SAPIENZA
UNIVERSITÀ DI ROMA

Integrating Transactional Memory Support in the TensorFlow Framework

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Master of Science in Engineering in Computer Science

Candidate

Davide Leoni
ID number 1614811

Thesis Advisor

Prof. Bruno Ciciani

Co-Advisors

Prof. Francesco Quaglia
Prof. Paolo Romano
Prof. Bruno Martins
Manuel Barreto Lima Reis

Academic Year 2016/2017

Integrating Transactional Memory Support in the TensorFlow Framework
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: davide.leoni90@gmail.com

Acknowledgments

The thesis work described in this document was conducted from May to September 2017 at the Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa (INESC-ID) [1] *within the SATURN* [2] *(Spatio-temporal cloud store for big-data applications) project. The work was done together with Manuel Barreto Lima Reis (who is currently pursuing a PhD degree in Computer Science) and was supervised by Prof. Paolo Romano, from Instituto Superior Técnico of the University of Lisbon, Prof. Bruno Emanuel da Graça Martins, from the same institute, and Prof. Francesco Quaglia, from Università degli Studi di Roma Tor Vergata. I want to thank all of them: collaborating with them was a pleasure and allowed me to improve much my technical preparation. I am really grateful to Paolo and Francesco, as they gave me the opportunity of working on a very interesting project, in a stimulating international environment. I enjoyed much my time in Lisbon, in particular at the INESC-ID institute, and I learned a lot, both from the professional and the personal point of view. I must acknowledge all the colleagues from the institute, who offered me a great support: Daniel Castro, Pedro Raminhas and Shady Alaa Issa. My sincere appreciation also goes to Bruno, for his contribution and for the time he dedicated to me. A special thank goes to my friend Manuel, for his fundamental help and for all the good moments spent together. Last, but not least, I must thank my advisor, for his availability and support.*

[1] www.inesc-id.pt

[2] www.gsd.inesc-id.pt/blog/saturn

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivations

It only takes to look in any academic search engine at the number of publications related to it, to realise how *machine learning* has been having a growing momentum in the last decades. Nowadays, the core business of the main world-wide leading companies, like Facebook, Google or Amazon, is largely based on the capacity of training models. This is particularly true for one of the branches of machine learning, *deep learning*, whose success has been impressively increasing in the very last few years at a large scale. Such a popularity is understandable considering the great progresses made in various areas (for example computer vision, natural language processing, speech recognition and information retrieval) thanks to the algorithms provided by this research field [1, 11, 27].
Hence it does not come as a surprise to know that many frameworks and libraries, both open and closed source, exist to implement and experiment with these algorithms. Nevertheless, although machine learning, and even more deep learning, have been only recently catching the attention of the researchers, it is been now many years since the first libraries and frameworks for machine learning have been introduced. A list of the most famous ones can include, in chronological order, the *OpenCV* library for C++, the *scikit-learn* library for Python, and the *Apache Hadoop* platform for Java. Focusing on deep learning, the oldest example, *Torch*, dates back to 2002 and was then followed by *Theano* and *Caffe*.
In November 2015, Google released *TensorFlow* (TF), a software library for the implementation and training of machine learning models. After the very first releases, whose performances were poor compared to the other libraries [4], TF rapidly gained a broad consensus among developers (as witnessed by the thousands of projects on GitHub making use of it) and researchers. This is probably due to one characteristic that makes TF unique compared to its competitors: flexibility [27]. As a matter of fact, TF supports *distributed execution* across various kinds of devices, with very different computational power, from mobile phones to GPUs. Such an heterogeneity of hardware platforms reflects into a variety of tasks and goals, which range from large-scale training with many dedicated machines in a production environment, to experimenting new algorithms on simple smartphones. Overall, on one hand, the objective pursued by TF is to make machine learning and deep learning accessible

to the broadest audience as possible; on the other hand, it puts a strong accent on the adoption of parallel and distributed computing to make machine learning faster [1, 2, 27]. Indeed, in the last decade the rapid spread of low-cost machines with multiple cores has made possible to apply machine learning to datasets much larger than before, allowing to improve the accuracy of trained models [6, 23]. Nevertheless, because of the synchronization primitives that have to be used when the training is performed in parallel, the scalability is still limited.

1.2 Goal

The great popularity of TF, together with the current emphasis on training models in parallel, suggested the idea of investigating the benefits of providing this framework with the support for *transactional memories*. These, indeed, removing the need for the classic synchronization primitives (more on this in Section 2.1), can help to improve the scalability of programs run on multiple processors. As a consequence, the algorithms provided by TF, when applied in the context of large-scale training, could potentially profit from the application of transactional memories. The intended goal of this thesis consisted in verifying whether this speculation was true.

Chapter 2

Background

This chapter contains an overview of the fundamental concepts which this dissertation is based on. The first part deals with the transactional memories, starting from the reasons that motivated their introduction, describing the general design concepts and bringing some concrete examples. The following section is dedicated to TensorFlow, the library for machine learning by Google, at the center of this thesis work. Considering its complexity, and the high number of functionalities, it is not trivial (and out of scope) to provide a complete description of the framework. The attention is thus focused on the aspects that are most relevant in the perspective of integration the transactional memory support. The last section provides a theoretical background related to the machine learning models taken into account as testbeds for the experiments reported in this work.

2.1 Transactional Memories

2.1.1 Introduction

After many years since the first parallel architectures came to light, it is now clear that writing programs that exploit effectively all their computational power is not a trivial task [14]. Indeed, the well-known synchronization primitives (locks, mutexes, semaphores, etc.), required to coordinate the action of threads (or processes) on shared data, introduce some complications. On one hand, it is necessary to keep as tiny as possible the *critical sections*, the piece of code protected by the primitives, in order to provide scalability. On the other hand, the finer the grain of the critical sections, the more complicated and the trickier to debug is the associated code. Besides performances, when not properly used synchronization primitives can threat the progress of the programs (*deadlocks*) or their correctness (*race conditions*). Last, but not least, explicit synchronization is not composable. In alternative, programs designed to run in parallel can adopt *lock-free algorithms*, but it is even more difficult to design them and to prove their correctness, so they are actually used for very specific domains.
Given all these issues, *Transactional memories* were introduced to provide programmers with a new abstraction that made parallel programming easier. The idea was borrowed from the world of databases, which are able to run queries simultaneously on multiple processors without the programmer to be concerned about

the concurrency of the execution: this is achieved wrapping the computation with *transactions*. In databases, transactions are the fundamental unit of execution: they are a set of instructions that have to be executed sequentially. They have four basics characteristics:

1. *atomicity*: the effects of a transaction are made visible (the transaction is said to have *committed*) only if all the enclosed operations completed successfully. If any of these fails, the whole transaction is said to have *failed* and the system remains untouched as if the transaction had been never executed
2. *consistency*: the transaction must not alter the consistency properties of the database
3. *isolation*: each transaction must not interfere with other transactions (if any) executed simultaneously
4. *durability*: once a transaction has committed, its effects on the database are definitive

Each transaction is executed as if it the associated computation was the only one running in the system, despite multiple transactions can be executed simultaneously. The underlying implementation deals with the concurrency issues transparently to the programmer and still guarantees *serializability*: the net result of the simultaneous execution of the transactions is the same as if the transactions run sequentially, so the programmer is not worried by synchronization of the operations. Transactional memories are intended to provide the support for performing the concurrent read and write to shared memory addresses applying the semantics of the database transactions [14]. As a consequence, programmers are relieved from the burden of explicitly applying synchronization primitives, since the properties of the transactions offer the guarantees necessary to solve the synchronization problems. Developers only have to identify the code associated with a critical section and surround it with a transaction using an interface similar to the one shown below.

```
int balance;
lock protect_balaance;

void deposit (int x) {
        protect_balance.lock();
        balance += x;
        protect_balance.unlock();
}

void withdraw (int x) {
        protect_balance.lock();
        balance -= x;
        protect_balance.unlock();
}
```

```
void deposit (int x) {
     startTx();
     balance += x;
     endTx();
}

void withdraw (int x) {
     startTx();
     balance -= x;
     endTx();
}
```

Listing 2.1. Locks vs. Transactions

2.1.2 Implementation

This section provides an overview of the underlying mechanisms of transactional memories, independently of their specific implementation. First of all, it is necessary to handle the concurrent access to shared data in order to achieve isolation between transactions. Two transactions are said to determine a *conflict* if they execute two conflicting operations (two writes or one read and one write) on the same memory addresses: not only have conflicts to be detected, but they also have to be resolved. Three types of approach are possible [14]:

- *pessimistic*: at the moment of executing each operation inside a transaction, it is checked whether it will not produce a conflict; if that is so, the system takes some actions to avoid it
- *optimistic*: transactions are executed until a conflict is detected at only at that the point, actions are taken to resolve the conflict, possibly aborting transactions
- *hybrid*: it is possible, for example, to apply the pessimistic control only to conflicts between write operations and use instead the optimistic control for read operations. Another possibility is to use optimistic control and define some *irrevocable transactions* which are guaranteed to complete

In case the optimistic concurrency control is chosen, a further design decision regards the granularity of the detection of a conflict: in *Software Transactional Memories* (see Section 2.1.3) the resolution of the detection corresponds to entire objects, while in *Hardware Transactional Memories* (see Section 2.1.4) corresponds to single lines of the cache. Another important aspect of the detection is the time when it occurs. There are three moments, in chronological order with respect to the execution of the transaction, when a conflict can be detected [14]:

1. before the transaction is actually started
2. during its execution, a transaction can check whether the data it has accessed have been modified by other concurrent transactions
3. at "commit-time', when the transaction has already concluded its work

Furthermore, in order to detect conflicts, some implementations monitor the intersection of the set of accessed addresses only between committed and on-going transactions, while others also consider the intersection between transactions that are not terminated yet. Finally, transactional memories are required to keep track of the effects produced by the operations they contain, in order for them to be removed should the transaction be aborted. As regards with this aspect, there are two alternatives [14]:

1. *direct update*: the write operations directly replace the content of memory addresses to which they are applied and the old content is stored in an *undo-log*: this will be consulted, in case of abort, to restore the state of the memory antecedent to the execution of the transaction
2. *deferred update*: the results of the write operations are written to a *redo-log* and if the transaction eventually commits they are propagated to the memory

2.1.3 Software Transactional Memories

The idea behind *Software Transactional Memories* was inspired by the work presented in [28], which first introduced the concept of "transaction" outside the domain of databases. Anyway, the current implementations of STMs are quite different from the mechanism presented in that work, where a transaction was required to first declare the memory addresses it was about to work with [14]. Many examples of STMs exist nowadays and each represents a different approach in the implementation of some basic mechanisms necessary to build STMs.
One of these mechanisms regard the storage of the *metadata* necessary to keep track of the memory locations accessed by the transactions. Two alternatives are possible:

1. storing the metadata as part of the object allocated by the program (*object-based*)

2. creating a mapping from each memory location to the address where are stored the related metadata (*word-based*)

The first solution is more convenient in terms of time required to access the metadata, since they are tightly connected to the object they refer to; on the other hand, since the metadata are usually stored in the same cache line as the object, two concurrent accesses to different fields of the same object will be detected as a conflict by the cache coherency protocol. Undo-logs (see Section 2.1.2) are usually allocated using lists, while for redo-logs the implementation has to be more complex in order to avoid the degradation of performances. Unlike the undo-log, indeed, the redo-log has to be checked every time a transaction reads a memory location previously written by another transaction, so that the former can see the speculative updates performed by the latter. The computational complexity of searching for these updates in a list is linear, and this is not acceptable from the point of view of performances, so often a hash table is used instead. STMs also need to keep track of the memory locations a transaction read from (*read-set*) and of those it wrote to (*write-set*).
The STM implementation used in some of the experiments described in this dissertation is *TinySTM* [1] [9, 10], which inherits many of its mechanism from the *TL2* algorithm [8]. This algorithm makes use of *versioned locks* to store the metadata relative to a memory location accessed by a transaction. Versioned locks combine a mutual exclusion lock, that has to be acquired by write operations, with a version number, incremented every time the lock is grabbed and then released. If the lock is free, no write operations is currently being performed and the version number reflects the current state of the object; otherwise, the version number refers to the transaction that is accessing the object the metadata is associated with. TL2 adopts deferred updates: the lock, that protects the critical section executed transactionally, is acquired only when the transaction is ready to commit. Moreover, TL2 relies on a *global clock*, namely a counter, unique for the whole application, incremented by one when a writing transaction commits. When a transaction starts, it reads the value of the global clock: this value is referred to as *read-version* to distinguish it from the *write-version*, which instead is the value of the global clock incremented by a transaction before committing. The version number of each object is equal to the

[1] tmware.org/tinystm

write version of the last transaction that modified it and successfully committed. A read-only transaction works as follows:

1. gets a read-version, reading the global clock
2. speculatively performs load operations: before each load, checks whether the versioned-lock of the memory address is free and whether the version number is less than or equal to the read version. If either of the two checks fails, the transaction is aborted, because this means that a write transaction meanwhile committed and modified the value of the location

It has to be noted that read-only transactions have the great advantage of avoiding that a read-set gets built for them; moreover, the read-version ensures the consistency of the view of the memory of a transaction. Unsurprisingly, a write transaction is more complex:

1. gets a read-version, reading the global clock
2. speculatively performs load ans store operations, logging them respectively in a read-set and a write-set. The load instructions, before actually accessing the memory, first check if the address is contained in the write-set and, if so, the corresponding value is returned. Moreover, for each load is made the same double check as in read-only transaction: this ensures that a memory location has not been concurrently modified by another transaction.ì
3. when the transaction is ready to commit, it tries to acquire all the locks of the locations in the write-set: if only one of these is not successfully grabbed, the transaction aborts
4. the write version is obtained, reading the global clock after having incremented it
5. before committing, the read-set is further validated to guarantee that, during the preceding two steps, no modification has been made to the addresses read
6. commit: for all the locations in the write set, the new value is stored, the version number is initialized to the write-version and then the lock is released

The main advantages of implementing transactional memories in software rather than using hardware resources is that software in general is easier to adapt and to modify.

2.1.4 Hardware Transactional Memories

Hardware Transactional Memories represent a step forward with respect to STMs under many points of view. From the performances point of view, for example, they have less overhead than STMs. Also they can be applied more transparently to existing environments and they allow to save energy compared to STMs. Analogously to STMs, various types of HTMs have been proposed, but it still possible to categorize them on the basis of some design choices. Firstly, as regards with the definition of the boundaries of a transaction, there are two types of HTMs:

1. *explicit HTMs* provide new sets of instructions to directly specify which load and store instructions have to be performed inside a transaction; examples are [15, 30]

2. *implicit HTMs* only allow to specify where a transaction starts and where it finishes, treating all the memory accesses in between as part of the transaction; [25] and [7] belong to this group

HTMs usually exploit the hardware cache to store the *read set* (set of memory addresses read by a transaction) and the *write set* (set of memory addresses modified by a transaction). For the read set this is as simple as setting an extra "read bit" in the cache line containing the address accessed by a transaction. As for the write set, this is not so easy to deal with, since some cache coherency protocols allow that a modification to an address is first written in the cache and then propagated to the main memory. An alternative approach consists in allocating dedicated extra spaces for the read and write sets. Nevertheless, the usage of the hardware cache has a further advantage: it allows to rely on the *cache coherency protocols* for the detection of conflicts. As a matter of fact, the most common of this protocols, *M.O.E.S.I* defines five states for a cache line (modified, owned, exclusive, shared and invalid) which allow to determine whether two transactions are accessing memory locations in a conflicting manner. As soon as a conflict is detected, the usual strategy adopted by HTMs consists in aborting the transaction and passing the control to a software handle to either use a fallback path or re-execute the transaction.
The hardware support for transactional memories is now a mature feature of processors: the next two sections provide a description of two commercial examples of HTMs that were used for this thesis.

2.1.5 POWER8

The *POWER8* processor, by IBM, features twelve cores, each having at disposal its own Level2(L2) cache, with a capacity of 512KB, and and Level3(L3) cache, capable of storing information up to 8MB. There are also two memory controllers (both third generation PCI Express) and a bus providing connection among all the components. Each core (64 bit) supports multithreading, being able to simultaneously run up to 8 threads, and is provided with a 32KB eight-way associative Level1(L1) cache for instructions and a 32KB eight-way associative Level1(L1) cache for data [12].

Transactional Memory Implementation

POWER8 is the first processor, based on POWER ISA, that supports transactional memory: the instruction set is extended with a set of machine instructions to implement and control the execution of transactions. The implementation is *best-effort*, namely there is no guarantee that a transaction will eventually commit. In case of failure, a dedicated register, the Transaction Exception and Status Register (TEXASR), is initialized to a code (eight different values are possible on POWER8) that gives an indication of what prevented the transaction from committing; another register, the Transaction Failure Instruction Address Register (TFIAR) is instead set to the address of the instruction responsible for the failure. More importantly,

if a transaction fails, a software handler is activated to take care of it: the usual behaviour of the handler consists in reading the value of the two registers and deciding, depending on the cause of the failure, whether it is worthy to try re-executing the transaction or if it is better to abort it. Transactions can be nested (up to 62 levels), but the results of the commit of a inner transactions are not made available until the outer transaction correctly finishes too; if a transaction, at any level of nesting, aborts, the whole stack of transactions aborts too. The programmer requests for the beginning of a transaction in a thread issuing the `tbegin` instruction. The Machine Status Register (MSR) is used to determine whether an execution path is running inside a transaction or not. These two conditions correspond to the *Transactional* and *Non-Transactional* state respectively; the third possible state is *Suspended* (see later). The begin instruction causes the underlying implementation to first make a checkpoint of the current state of the hardware registers; then it checks whether other threads are concurrently executing the operations on the same shared data outside the scope of a transaction. If that's so, the transaction can not proceed and it is aborted by the `tabort` instruction, which gives control to the software handler. Otherwise, the transaction can continue till the end. During the execution, write operations to memory are performed *speculatively*, namely their effect are made visible only after the transaction eventually commits, otherwise they are cancelled. If two different threads access the same cache line and either one or both are currently running a transaction, a conflict arises, causing either of the transactions to fail. The size of a cache line determines the capability of the implementation of detecting *false conflicts* (those situations where two threads are actually accessing different memory locations but these operations are still recognized as conflicting because the two addresses are stored in the same cache line). In case of POWER8, the cache line has a size of 128 bytes [24]. The programmer can also temporarily suspend a transaction with the `tsuspend` instruction: from that point on, the instructions will not be included in the transaction, so their effects will not be undone should the transaction later fail; indeed, the cache lines used by the transactions before it was suspended keep being monitored, so if a conflict takes place the transaction will fail as soon as it is resumed with the `tresume` instruction. The possibility of suspending transactions serves for debugging purposes and for the transaction itself to continue after an interrupt has occurred. The end of a transaction is marked by the programmer with the instruction `tend`: if the transaction can successfully commit, its associated write operations are atomically committed for all the other threads to see them; the effects of multiple transactions are applied following a serial order. Otherwise, if the transaction can't commit, the software handler usually tries to re-run it for a limited number of times; if it never succeeds, the code of the transaction is executed acquiring the lock that protects the critical section [19].

Transactional Memory Architecture

Fig. 2.1 (the image taken from [19]) provides a clear illustration of the internal hardware architecture of POWER8: what follows gives a brief explanation of its most important elements. The Instruction Sequencing Unit (ISU) stores in the core the checkpoint of the registers created every time a new transaction begins. The Load Store Unit (LSU) of each core contains a Level1 private cache where are

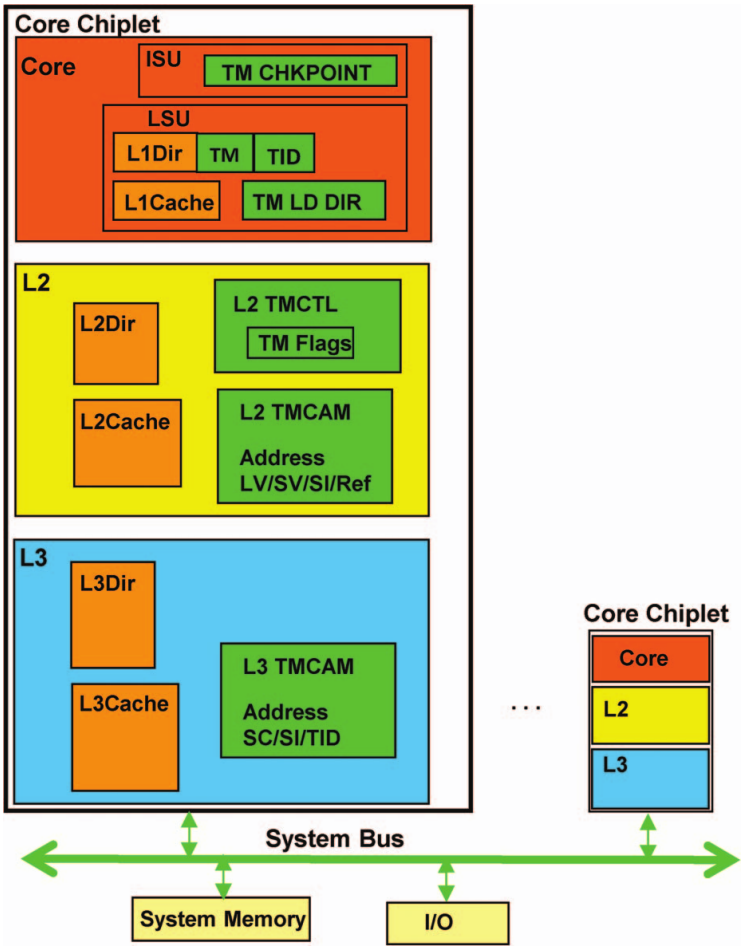

Figure 2.1. The HTM architecture on POWER8

stored the addresses written by the transaction. In particular, for each cache line the L1 Directory (L1Dir) includes two extra bits that are set to respectively indicate whether the cache line has been accessed inside a transaction and the ID of the thread executing it. Cache lines involved in a transaction are not available to other threads until the commit or the abort of the transaction itself: any attempt of access is translated to a cache miss and handled by L2 cache. The TM Load Directory (TM LD DIR) keeps a reference to the addresses read by transaction, which are maintained in the L2 cache, and serves to reduce transfers to the L1 cache. In the L2 cache, the TMCAM is a directory with 64 entries, each containing a cache line address and several bits that indicate its state for each single thread. Conflicts are detected by the TMCTL unit of the L2 cache, which keeps an eye on the memory locations accessed by not only the local core, but also by the other cores. When a conflict actually occurs, the core running the aborted transaction gets informed by the TMCTL unit and the software handler is activated. Finally, the TMCAM in the L3 cache keeps track of dirty cache lines accessed by a transaction during its execution, until it eventually fails or commits.
The size of the described hardware resources, necessary for the hardware implemen-

tation of the transactional memory, puts a limit on the total number of memory addresses that can be accessed inside a transaction. When a transaction tries to overcome this limit, the transaction is aborted: such a kind of abort is referred to as *capacity-overflow abort* [24].

2.1.6 Intel Haswell

Haswell [13] is the fourth-generation family of Intel Core processors, by Intel. According to jargon used by Intel to classify the advancements in the architectures of CPUs, it is a "tock", meaning that it represents a big step with respect to the microarchitecture of its predecessor, Ivy Bridge. Despite the main improvements affect the area of power management and efficiency, Haswell features, for the first time in the history of Intel architectures, a new set of instructions taht provide support for transactional memories: this extension of the ISA is called *Transactional Synchronization Extensions (TSX)* (see Section 2.2.1).
Whereas plenty of information is available regarding the interface exposed to programmer for them to take advantage of the transactional memory support, it is way more difficult to find details about its underlying implementation, since they were not disclosed. It is sure [13] that the Level1 cache, whose capacity is 32KB, serves to maintain the memory addresses accessed by the transaction during its execution, for both load and store instructions. Moreover, conflict detection relies on cache coherency protocols. Some [2] speculated about the rest of the implementation, formulating the hypothesis that write operations are first executed on cache lines, saving the original data present in memory in a shared L3 cache.

2.2 Transactional Lock Elision

Transactional lock elision (TLE) is a technique that exploits hardware transactional memories to execute simultaneously on multiple threads critical sections that otherwise would be serialized by a synchronization primitive, running sequentially. The programmer is required to define the boundaries of the critical section to be run in parallel and the processor, at runtime, avoids taking the lock which protects the critical section and executes it in parallel on multiple threads in a "transactional way". This means that the processor does not actually execute the write operation directly to memory, but buffers them waiting for the outcome of the transaction; analogously, it tracks all the memory addresses read. At the end of the transaction, if there was no conflict in the access of shared data by concurrent threads, the processor atomically applies the buffered store operations to memory. If that is so, the lock has been "elided", since the critical sections have been executed in parallel without synchronizing the access to shared data and still ensuring isolation, consistency, durability and atomicity (the properties inherited from the transactional semantics). Otherwise, if the transaction fails, the state of the system is restored to the moment antecedent to its start. The next two sections contain a description of the interface offered to the programmer for the definition of transactions by the two architectures described before.

[2] realworldtech.com/haswell-tm

2.2.1 Intel Transactional Synchronization Extensions

Starting from the family of processors Haswell (see Section 2.1.6), Intel adopted a new instruction set, Intel TSX, that comprises two software interfaces for programmers to request transactional lock elision:

1. *Hardware Lock Elision (HLE)*
2. *Restricted Restricted Transactional Memory (RTM)*

The underlying implementation is best-effort, meaning that transactions are not guaranteed to always commit, even in absence of concurrency, because of conflicts or capacity aborts. Another reason for failures is the execution of machine level instructions that are guaranteed to cause transactions to abort, like `CPUID` or `PAUSE` [3]. It is up to the programmer to ensure that these kind of instructions are not included in a transaction, as well as to try avoiding the following (partial) list of instructions, which might may cause an abort too:

- system calls (`SYSENTER, SYSCALL, SYSEXIT, SYSRET`)
- interrupts (`INTn, INTO.`)
- I/O instructions
- updates to some portions of the `EFLAGS` register
- saving the state of the processor (`XSAVE`)

HLE is an extension of the machine instructions set that introduces two new instruction prefixes, `XACQUIRE` and `XRELEASE`, to mark respectively the beginning and the end of a transaction. The two prefixes are ignored on legacy architectures that do not have the transactional support, hence HLE is the recommended choice for programmers that intend to run their code on the broadest set of architecture as possible. The mechanism associated with HLE tries to run the code comprised between the two prefixed instructions as a transaction: in case of abort, there is no further attempt to make the transaction commit.
RTM, on the contrary, although relies on the same hardware resources, is capable of offering the developers more flexibility in the management of the transactions. It is a new set of instructions that a programmer has to use in his code to handle transactions:

- `XBEGIN` serves to define the beginning of a transaction and takes in input the address of a *fallback function*, which is passed the control of the execution if the transaction does not successfully commit. One common behaviour for this software handler is to try re-executing the transaction a number of times: in case all the attempts are not successful, the critical section is executed in a non-transactional way grabbing the associated lock
- `XEND` marks the end of the transaction

[3] software.intel.com/en-us/node/524022

- `XABORT` allows to explicitly abort an ongoing transaction, also specifying the motivation of this action through an 8-bit value

```
1  retries = N_RETRIES
2  status = XBEGIN()
3  lock = init_lock()
4  if status != ok:
5      if retries == 0:
6          lock.acquire()
7      else:
8          retries --
9          goto 2
10 if islocked(lock) == true:
11     XABORT
12
13 CRITICAL SECTION
14
15 if attempts == 0:
16     lock.release()
17 else
18     XEND
```

Listing 2.2. Transactional Lock Elision with TSX

In case of abort of a transaction, even due to a call to `XABORT`, the implementation sets the bits of the `EAX` register in order to indicate the reason of the failure. The following is an explanation of the meaning associated with each bit of the register in case of abort:

- bit 0: set if `XABORT` was called
- bit 1: set if a re-execution of the transaction might commit successfully; cleared if bit 0 is set
- bit 2: set in case of conflicting access to shared data
- bit 3: set in case of buffer overflow
- bit 4: set in case of debugging point being hit
- bit 5: set in case of abort of a nested transaction
- bits 24 to 31: contain the 8-bit value passed by `XABORT`

2.2.2 POWER ISA Transactional Memory Support

As already anticipated in 2.1.5, POWER 8 is the first family of microarchectures manufactured by IBM to have the transactional memory support. The new instruction set has an extension that, similarly to Intel, allows programmers to run a critical section speculatively, without first acquiring the lock and possibly bypassing (or

"eliding") it. The boundaries of the code to be executed speculatively are defined by the programmer with the machine level instruction `tbegin` and `tend`; a running transaction can be momentarily suspended with the `tsuspend` instruction and later restored with the `tresume` instruction. Like Intel TSX, also the POWER ISA implementation is best-effort, so when a transaction starts, a checkpoint of the actual state of the registers (so called *pre-transactional state*) is automatically created: in case of a premature interruption of the transaction, the condition of the system is rolled-back to this initial state. Privileged users can access the values of the checkpointed registers by mean of the `treclaim` instruction; conversely, the `trechkpt` instruction places the checkpoint back to the dedicated registers. If a transaction fails (either because of external factors or because of an explicit termination requested by the programmer through `tabort`), the control has to be redirected to a software handler. In particular, the `CR0` register is initialized to the value `0b1010` in case the transaction cannot be successfully start, to the value `0b0100` otherwise. As a consequence, the instruction following `tbegin` has to be a branching instruction, for example `beq` (branch on equal), that checks the value of the `CR0` register and gives control to a software handler in case of failure.

```
1  start_transaction:
2    tbegin                    # Start transaction
3    beq     software_handler  # Branch on value of CR0
4    CRITICAL SECTION          # Critical section
5    tend.                     # End transaction
6    b       transaction_exit  # Continue execution after
7                              # a transaction has terminated
8  software_handler:
9    mfspr   r4, TEXASRU       # Read the value of
10                             # the TEXASR register
11   andis. r5, r4, 0x0100     # Check if the failure
12                             # is persistent
13   bne     acquire_lock      # If it is persistent,
14                             # acquire the lock
15                             # to run the critical section
16   b       start_transaction # If not persistent, try again
17
18 acquire_lock:
19
20 transaction_exit:
```

Listing 2.3. Transactional Lock Elision with POWER ISA

A failure is either *transient* or *persistent*: in the former case, the abort of the transaction was caused by a temporary condition of the system, so there are chances that the transaction commits on a new attempt; in the latter case, instead, there are no possibilities for a successful completion of the transaction, so the strategy adopted in this case consists in acquire the lock and run the critical section in non-transactional way. The software handler leverages the content of the `TEXASR` (see Section 2.1.5) register to determine the cause of the failure. More precisely, the

first 6 bits of this 32 bit register contain the transaction failure code; the seventh bit is set in case of persistent failure [12].

2.2.3 GCC Transactional Memory Interface

On machines based on POWER ISA, like POWER 8, developers can take advantage of the transactional memory facility through *GCC (GNU Compiler Collection)*. There are three possible choices [12]:

1. using the *HTM low-level built-in functions*
2. using the *HTM high-level built-in functions*
3. using the *libitm* library

The first set of functions is enabled compiling the source code with the `-mcpu=power8` or `-mthm` options. All these functions return the 4-bit value written by the corresponding machine level instructions in the `TEXASR` register; the header file `htmintrin.h` includes some helper functions to parse and return this value. The macro that starts a new transaction, `__builtin_tbegin` does not adhere to this procedure and returns instead a boolean value indicating whether a transaction has been successfully started. The listing below shows a pattern of usage of the functions.

```
1  #include <htmintrin.h>
2
3  int num_retries = 10;
4
5  while (1)
6    {
7      if (__builtin_tbegin (0))
8        {
9          /* Transaction State Initiated.  */
10         if (is_locked (lock))
11           __builtin_tabort (0);
12         ... transaction code...
13         __builtin_tend (0);
14         break;
15       }
16     else
17       {
18         /* Transaction State Failed.  Use locks if the
               transaction
19             failure is "persistent" or we've tried too many
                   times.  */
20         if (num_retries-- <= 0
21             || _TEXASRU_FAILURE_PERSISTENT (
                   __builtin_get_texasru ()))
22           {
23             acquire_lock (lock);
```

```
24              ... non transactional fallback path...
25              release_lock (lock);
26              break;
27            }
28        }
29    }
```

Listing 2.4. Low-level HTM GCC built-in functions on POWER ISA

The code above [4] tries to execute speculatively the code corresponding to a critical section (`transaction code`) until either the associated transaction (identified by the integer passed to `__builtin_tbegin`) commits or a limit for the number of attempts (`num_retries`) is reached. As soon as the transaction starts, the code checks whether other threads are concurrently executing the same critical section acquiring the lock that protects it (see Section 2.1.5). If that is so, the transaction is explicitly aborted by `__builtin_tabort`; otherwise it proceeds until the end of the critical section, which is marked by `__builtin_tend`. In case of failure of the transaction, the software handler (corresponding to the `else` branch) reads the value of the `TEXASR` register with `__builtin_get_texasru` and thanks to the utility function `_TEXASRU_FAILURE_PERSISTENT` determines if it is worthy to try re-executing the transaction. If this is not so, the fallback path, namely the lock-protected execution of the critical section, is taken.
The second group of functions is intended to make the code more portable, since it can be compiled not only by GCC, but also by IBM XL; programmers have to include the header `htmxlintrin.h` to use this interface.
Finally, the highest level interface available is represented by *libitm*, the *GNU Transactional Memory Library*[5]. At runtime, it transparently verifies whether the underlying hardware supports transactional memory and, if so, wraps a transaction around the critical section specified by the programmer as in the listing below. The integration of this

```
1  __transaction_atomic {
2    CRITICAL_SECTION;
3  }
```

Listing 2.5. Transactional code using *libitm*

GCC also offers support for programmers to use hardware transactional memories on machines based on Intel TSX. The RTM interface (see Section 2.2.1) can be accessed adding `-mrtm` to the flags passed to the compiler and comprises four functions (see the code [6] below:

- `xbegin` starts a transaction. If this operation is successful, `_XBEGIN_STARTED` is returned; if the transaction is aborted, besides undoing its effects, this function stores in the `EAX register` a code indicating the reason for the abort and

[4] gcc.gnu.org/onlinedocs/gcc/PowerPC-Hardware-Transactional-Memory-Built-in-Functions.html
[5] gcc.gnu.org/wiki/TransactionalMemory
[6] gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/X86-transactional-memory-intrinsics.html

returns it. This provides the software handler with the information necessary to decide to either retry the transaction or not

- `xend` commits a transaction. If this operation is successful, the effects of the operations inside the transaction are applied atomically, otherwise the state of the registers is restored to the state antecedent to the transaction
- `xabort` explicitly aborts a transaction (including nested transactions); the constant to be passed as parameter is written into the upper eight bits of the `EAX` register
- `xtest` serves to check if a transaction is currently active: 0 is returned if that's not the case

```
1  #include <immintrin.h>
2
3  int n_tries, max_tries;
4  unsigned status = _XABORT_EXPLICIT;
5  ...
6
7  for (n_tries = 0; n_tries < max_tries; n_tries++)
8    {
9      status = _xbegin ();
10     if (status == _XBEGIN_STARTED || !(status &
         _XABORT_RETRY))
11       break;
12   }
13 if (status == _XBEGIN_STARTED)
14   {
15     ... transaction code...
16     _xend ();
17   }
18 else
19   {
20     ... non-transactional fallback path...
21   }
```

Listing 2.6. Transactional code using the GCC RTM interface

The HLE interface (see Section 2.2.1) is accessible through the atomic built-in functions, which allow to synchronize the access to memory locations by multiple threads in such a way that read and write are made visible atomically. A transaction is started by the following snippet of code [7]

```
1  /* Acquire lock with lock elision */
2  while (__atomic_exchange_n(&lockvar, 1, __ATOMIC_ACQUIRE|
     __ATOMIC_HLE_ACQUIRE))
```

[7] gcc.gnu.org/onlinedocs/gcc/x86-specific-memory-model-extensions-for-transactional-memory.html

```
3     _mm_pause(); /* Abort failed transaction */
```

Listing 2.7. Start transaction with HLE

and finished by this piece of code [7]

```
1 /* Free lock with lock elision */
2 __atomic_store_n(&lockvar, 0, __ATOMIC_RELEASE|
    __ATOMIC_HLE_RELEASE);
```

Listing 2.8. End transaction with HLE

2.3 TensorFlow

2.3.1 Introduction

Ever since it was released by Google in 2015, TensorFlow (TF) has become one of the most successful open-source software library for building and training machine learning models, especially deep-neural networks [11]. TF is the successor of DistBelief [2], a system for large-scale deep learning developed within the Google Brain project for internal usage: in particular it was meant to support research and to implement some of the functions of Google products like Google Maps or YouTube. Besides the fact that DistBelief is not open-source, TF was introduced to overcome the limits of its predecessor[1]. In particular, TF is intended to ease the definition of new architectures for the layers of the neural networks, as well as the experimentation of various optimizations for the training algorithms. Furthermore it allows to model further kinds of neural networks, like recurrent neural networks. More importantly, TF is not only capable of scaling-up to train on big amount of data, like DistBelief, but it can also scale-down to allow training networks even with limited hardware resources at disposal. As a result, TF can be run on a plethora of heterogeneous platforms, ranging from mobile devices to large-scale training systems featuring many GPUs.
The core of the library is implemented in C++ and exposes an API for the following programming languages: Python, C++, Java and Go; actually the first one is the recommended and most complete API. The logical organization of a client program that makes use of TF comprises two sections [8]:

1. a dataflow graph is built in order to represent the computation and the state associated with the desired machine learning algorithm

2. an optimized version of the dataflow graph is executed on the devices specified in the code

Postponing the execution, this can be optimized exploiting the information provided by the representation of the computation [1].

[8] tensorflow.org/get_started/

2.3.2 Building the Dataflow Graph

Following the dataflow programming model [29], a TF program is internally modelled as a directed acyclic graph where the nodes correspond to the computation to be performed; the edges carry the input data to the nodes and gather the resulting output data. The choice of such paradigm is motivated by the following advantages [9]:

1. the explicit connections between the nodes enable to rapidly determine the operations that can be executed in parallel
2. the computation can be distributed across multiple devices
3. it is possible to apply some linear algebra optimizations during the compilation and produce a faster code
4. the dataflow can be expressed in various programming languages, so it is portable

In TF each node of the graph is referred to as *operation* and represents the fundamental unit of computation: it is assigned a unique name and has a non-negative number of input and output data [1]. Each operation is executed on a *device*([[2]), like a CPU or a GPU, characterized by a type and a name. A *kernel* is the implementation of an operation optimized for a certain device, so for a single operation there are as many as kernels as the number of devices where the operation can be performed. Most of readily available kernels of TF rely on the *Eigen library*, which exploits C++ template to parallelize the execution of the code. Developers can add their own operations and related kernels into the library via a simple registration mechanism [1].
As the name suggests, in TF data flow along the edges under the form of *tensors*, namely *n-dimensional* arrays characterized by a primitive data type (integer, float, string) and a shape (the number of elements in each dimension) [10]. Training models usually implies multiple executions of the same dataflow graph: the lifetime of a tensor coincides with the duration of a single iteration and it maintains the same value throughout it. An exception to this rule regards the tensors returned by *variables* [11]. A variable is a special kind of operation, with no input, holding an internal buffer where values can be stored, modified and retrieved. Since variables survive a single execution of the graph, they are used to maintain the parameters of the models learned iteration after iteration. Another special kind of operation is the *queues*: like variables, queues persist values across successive executions using an internal mutable buffer, but unlike the former they implement the logic of a queue data structure.

2.3.3 Executing the Dataflow Graph

After having constructed the dataflow graph, it has no utility if it is not executed, because it is during the execution that the nodes are processed and produce a

[9] tensorflow.org/programmers_guide/graphs
[10] tensorflow.org/programmers_guide/tensors
[11] tensorflow.org/programmers_guide/variables

result. A *Session* object is the interface provided by the API for a client program to request the execution of the graph to the TF runtime, in particular invoking its *Run* method [1, 2]. This method has to be given a list with one or more *fetches*, namely the operations that have to be executed or the tensors that have to be evaluated. The C++ back-end automatically determines the subset of the nodes of the dataflow graph that have to be considered in order to satisfy the request and also the order in which they have to evaluated. This subgraph comprises the operation nodes in the provided list and all the other nodes whose outputs determine the inputs to the fetches. The user can control the execution providing input data under the form of *feeds*, i.e. a mapping between tensors present in the graph and the values to be assigned to them; typically these tensors are the *placeholders*, nodes through which a client program can feed data into the graph. Every execution of the Session through its Run method represents a *step* of the computation and TF allows concurrent executions of the same graph. By default there is no coordination between concurrent executions, which makes it possible to implement training algorithms where consistency is not required, but synchronization primitives are also provided when more guarantees are needed.

2.3.4 Visualizing the Dataflow Graph

The number of nodes of the dataflow graphs associated to some complex machine learning models can be over a few thousands, so it might be tricky trying to check how the computation is performed. In TF this problem is solved by *TensorBoard*, a tool that manages to provide a simplified yet complete and interactive visualization the dataflow graph created by a client program [2]. TensorBoard also allows to monitor many other metrics regarding the training of the model, like the value of the loss function or the values of the weights. To realize all this, the tool extracts all the information from special events files produced during the execution of the dataflow graph thanks to some *summary operations*. These special nodes have to be added to the graph by the developer and connected to those other nodes from which he/she wants to collect the information to be later visualized. When the dataflow graph is run, the summary operations generate the events file needed by TensorBoard and the graphical representation of the computation gets accessible at port 6006 of the local host. Below are a piece of code that computes the sum between the matrices in TF and the visualization of the corresponding dataflow graph in TensorBoard.

```
1  import tensorflow as tf
2
3  # the placeholder for the input matrices
4  A = tf.placeholder(tf.int32, [2, 2], "A")
5  B = tf.placeholder(tf.int32, [2, 2], "B")
6
7  # the Operation node that computes the sum of the two
       matrices and
8  # returns the resulting matrix
9  C = tf.add(A, B)
10
11 # the Summary operation to visualize the graph
```

```
12 summary = tf.summary.tensor_summary("Sum", C)
13
14 # create a Session using the context manager
15 with tf.Session() as sess:
16
17     # run the graph: provide a dictionary of values to
           initialize the input
18     # matrices and compute the sum of the matrices.
           Serialize the summary data
19     # into the "summary" protobuf object
20     summary, _ = sess.run([summary, C], {A: [[0, 1], [1,
           0]], B: [[1, 1], [0, 1]]})
21
22     # instantiate a FileWriter to write the serialized
           object to the disk
23     writer = tf.summary.FileWriter('.', sess.graph)
24
25     # write the serialized object to the disk
26     writer.add_summary(summary)
```

Listing 2.9. Add matrices in TensorFlow

2.3.5 Architecture

As soon as it has created the dataflow graph, the client program instantiates a Session, which serializes the graph and passes it to the core runtime of TF through an intermediate layer represented by a C API. The runtime of TF is composed of two main components:

- *distributed master*
- *worker service(s)*

The distributed master extracts from the entire dataflow graph the subset of nodes whose execution is necessary to evaluate the nodes indicated in the Session. The resulting portion of the dataflow graph is partitioned into one or more subgraphs, which are optimized and cached to make successive training steps quicker [1, 2]. The distributed master then schedules the execution of the subgraphs through a set of *tasks*, each running as a single operating system process. A task is responsible for the access to one or more devices and is associated with a *worker service* (a.k.a. *dataflow executor*). This receives the subgraphs from the master and decides the execution of the kernels corresponding to the operations across the set of available devices; when possible, kernels are executed in parallel (for instance using multiple CPU cores).
Two implementations of the above described architecture are available [2] (image was taken from [12]):

[12] tensorflow.org/extend/architecture

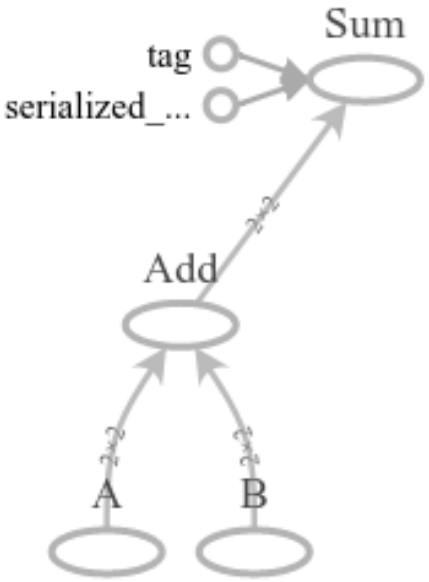

Figure 2.2. Dataflow graph relative to Listing 2.9

- *local*: the client, the master and one worker service all reside in the same process on a single machine
- *distributed*: the client, the master and multiple workers belong to the different processes on different machines connected over the network

In both cases, there are two possible scenarios:

- *single-device execution*
- *multiple-devices execution*

When only one device is available, a single worker service schedules the execution, for each operation node, of the kernel corresponding to the present device. If more the one device is eligible for running the nodes of the computation graph, the runtime of TF has to decide where to place the nodes and also hat to handle the communication between the devices in order to coordinate the execution. The algorithm adopted to determine the placement of the nodes takes into account parameters like the size of input and output tensors of a node and an estimation of the computation time required. After a simulation of the execution of the graph, the algorithm assigns each node to a device. This causes a partitioning of the original graph into as many

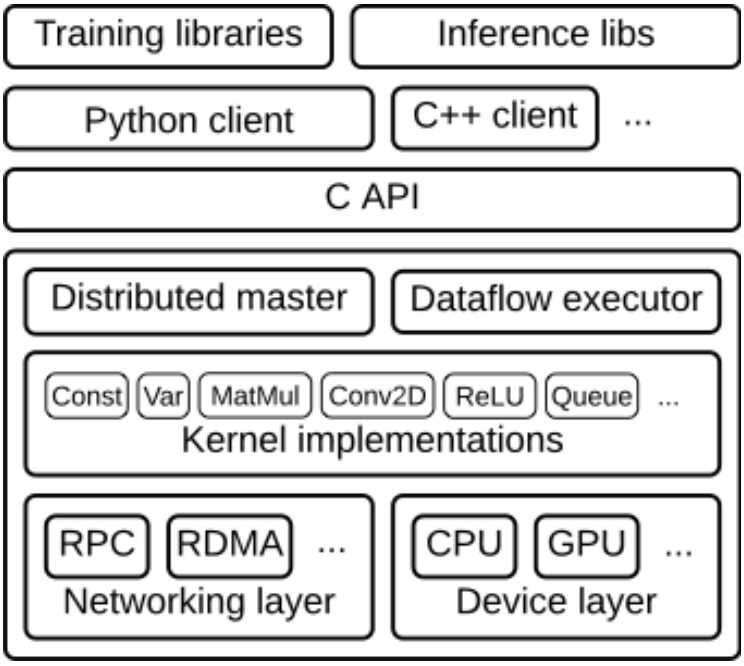

Figure 2.3. Architecture of TensorFlow

subgraphs as the number of devices. The starting and ending vertices of each edge of the original graph connecting two nodes assigned to different devices are respectively replaced by two new nodes *Send* and *Receive*. In this way, after the partitioning, those tensors that flowed along the replaced edge are transferred from one device to another. The actual realization of the two communication operations depends on the involved devices [1]:

- local transfers between a CPU and a GPU use the `cudaMemcpyAsync()` API
- local transfers between GPUs rely on *Direct Memory Access*
- transfers over the network between worker services adopt various protocols, mostly *gRemoteProcedureCall(gRPC)* and *RDMA over Converged Ethernet*

2.4 Statistical Learning

2.4.1 Overview

This area [16] deals with the solution of problems where, given a set of *input variables* $X_1, X_2...X_n$ and an *output variable* (or *dependent variable*) Y, it has to be found an estimation of the function f such that $Y = f(X) + \epsilon$, being ϵ an error term. The estimated function serves to predict the output corresponding to a certain input(s) or to understand the relation between input and output variables. Statistical learning requires the *training data*, namely a set of pairs $(x_i, y_i), i = 1...m$ where x_i and y_i are respectively the value(s) of the i-th observation for the input variable(s) and the corresponding value of the output variable. The estimation of the function f is

obtained through the application of a *learning method* to the these data. There are two main kinds of learning methods:

- *parametric methods*: they assume that the function f is approximated by a parametric model so the estimation of f is made finding the parameters of the models that minimize the distance between the function and the model themselves
- *non-parametric methods*: no assumption is made about the shape of f, so these methods usually achieve a better accuracy in the estimation at the cost of a higher number of observations.

So far we considered training data where for each value of the input variables observed the corresponding value of the output variable is known: in this scenario the training of the model is said to be *supervised*. Unfortunately it is sometimes the case that the values of the output variable are not known, which makes the training more difficult. The methods applied in this setting perform an *unsupervised learning*. Examples of the supervised learning are *liner and logistic regression*, while examples of unsupervised learning are *clustering methods*. A further classification of the problems tackled by statistical learning regards the kind of the variables involved: the domain of *quantitative variables* is numerical, while the domain of *qualitative variables* is a set of *classes*. Problems dealing with quantitative variables are referred to as *regression problems*, while those related to qualitative variables are known as *classification problems*. Given the variety of the learning methods available, a metric to evaluate their performances is needed in order to decide which of them achieve the best results for a certain dataset.
In case of regression problems, the chosen metric is usually the *mean squared error (MSE)* and it is calculated over the *test data*. This is a set of pairs (x_i, y_i), where x_i is the observed value(s) of the input variable(s) and y_i is the related value of the output variable, that is disjoint from the training set. Indeed it is not important that model performs well on the instances of the input values used for the training, but rather on unseen instances, because this ensures that it can predict the output with good accuracy for a future usage. Being m the number of test data pairs available and $\hat{f}$ the trained model, i.e. the estimation of the function f, the *test MSE* is defined as follows:

$$MSE_{test} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{f}(x_i))^2$$

Obviously, the smaller the test MSE, the better is the performance of the learning method. A risk associated with learning methods is *overfitting the data*, namely training the model in such a way that it is tightly connected to the training set. The negative consequence is that even though the model is accurate on training data, it does not perform well on test data.
As regards with classification problems, where the estimated function f determines the class which the input value belongs to, the measurement of the performances of the learning methods is based on the number of wrong classifications. Given $\hat{y_0}$ and

y_0 respectively the estimated class and the actual class (or *label*) of the input value x_0, the *indicator variable* I returns 1 if $\hat{y_0} \neq y_0$ and 0 otherwise. In other words, I returns 1 when a value is misclassified. Using the indication variable, it is possible to evaluate a classification algorithm through the *test error*, which is defined as the average value of the indicator variable over the test data. To put it simply, it coincides with the number of misclassifications over the number of test data, so the smaller is this value, the better is the classifier.

2.4.2 Linear Regression

Linear regression is a supervised learning method to predict the quantitative output variable Y given an input variable X, assuming the existence of a linear relation between them. In *simple linear regression* Y is approximated by the model $\beta_0 + \beta_1 X$: the training phase returns the estimation of its parameters, respectively $\hat{\beta}_0$ and $\hat{\beta}_1$. The most common approach for the estimation of the parameters is by far the minimization of the *least squares*. For each value $\hat{y_i}$ of the output variable predicted by the model as function of the input value x_i, a *residual* e_i is defined as the difference between the actual value y_i corresponding to x_i and the predicted value. The *residual sum squares (RSS)* is the sum of the squares of the residuals of all the values of the input variable. Applying some calculations it is easy to show that the values of the parameters of the model that minimize the RSS are the following (considering m possible values for the input variable):

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \beta_1 \bar{x}$$

with $\bar{y}$ and $\bar{x}$ the average value of the output and input variable respectively. When there are more than one input variables, simple linear regression is extended to *multiple linear regression*. Here the relation between the output variable and the n input variables is modelled by the equation $y = \beta_0 + \beta_1 X_1 + ...\beta_n X_n + \epsilon$. Also in this case, an estimation of the parameters $\hat{\beta}_0, \hat{\beta}_1 ... \hat{\beta}_n$ is obtained minimizing the RSS.

2.4.3 Logistic Regression

In a classification scenario, where the output response is qualitative (a class or a category), linear regression is to be replaced by *logistic regression*, which takes into account the probability that the output corresponding to a set of input values falls into a certain class. On the basis of such probability, it is possible to estimate the category which the input data belong to. For example, in case of binary class, given $p_1(X) = Pr(Y = 1|X)$, the estimated class of X is 1 if $p_1(X) > 0.5$, 0 otherwise. The linear model is no longer suitable to express the relation between input and output variables: indeed, because of the introduction of probability, there is the need of a function resembling a density function, whose response is comprised in the interval $[0, 1]$. The approach of logistic regression consists in modelling the likelihood of a certain class k conditioned to the input values using a *logistic function* (or *logistic sigmoid*) $p_k(X)$ ([[3, 16]), with $p_k(X)$ defined as follows:

$$p_k(X) = Pr(Y = k|X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}} = \frac{1}{1 + e^{-\beta_0-\beta_1 X}}$$

Given the assumption of a binary class, the input corresponds to either of the two possible values of the class, so it holds $p_0(X) = Pr(Y = 0|X) = 1 - Pr(Y = 1|X) = 1 - p_1(X)$. This can be expressed more concisely as

$$p(y|X) = p(X)^y + (1 - p(X))^{1-y}, y \in \{0, 1\}$$

which corresponds to the density function of a Bernoulli distribution. In this case, fitting the model means finding those values $\hat{\beta}_0$ and $\hat{\beta}_1$ such that, for each input x_i of the training data, $\hat{p}(x_i)$ it holds:

$$\hat{p}(y|x_i) = \Big(\frac{1}{1 + e^{-\hat{\beta}_0-\hat{\beta}_1 x_i}}\Big)^y + \Big(1 - \frac{1}{1 + e^{-\hat{\beta}_0-\hat{\beta}_1 x_i}}\Big)^{1-y}$$

is close to 1 when the class y_i of x_i is equal to y and 0 otherwise. One way to obtain an estimation is maximizing the *likelihood function*

$$l(\beta_0, \beta_1) = \prod_{i=1}^{m} \hat{p}(y_i|x_i) = \hat{p}(x_i)^{y_i} + (1 - \hat{p}(x_i))^{1-y_i}, y_i \in \{0, 1\}$$

assuming the independence of the m training examples (x_i, y_i). The maximization of the likelihood function is equivalent to the minimization of the *error* (or *loss*) function [3] defined as

$$E = -\log(l(\beta_0, \beta_1))$$

Applying the properties of logarithms, it easy to see that E is equivalent to the *cross-entropy* between the actual probability distribution of the class conditioned to the input, and its estimation:

$$E = \sum_{i=1}^{m} y_i \log(\hat{p}(x_i)) + (1 - y_i) \log(1 - \hat{p}(x_i))$$

Considering the non-linearity of the loss function, its minimization can be achieved appyling the *gradient descent algorithm* (see Section 2.4.4). Applying a similar reasoning, logistic regression can be easily extended to a scenario where there are $p > 2$ different classes. In this case, for each class k, $p_k(X)$ is modelled with a *softmax* function:

$$p_k(X) = Pr(Y = k|X) = \frac{e^{\beta_k X}}{\sum_{i=1}^{p} e^{\beta_i X}}$$

The name of this function derives from the fact that if there is a term $e^{\beta_j X}$, for some class j, which is far bigger than the other terms, this causes $p_j(X)$ to be almost equal to 1 and all the other probabilities to be almost zero. As a consequence, applying the softmax is like taking the "maximum" among all the $p_k(X), k \in \{1...p\}$; also it is guaranteed that $\sum_{i=1}^{p} p_k(X) = 1$. Fitting implies the minimization of the cross-entropy function with this model too.

2.4.4 Gradient Descent

Being β the set of parameters of a model, and $E(\beta|X)$ the error committed applying the model to a training set X, the goal of learning algorithms is minimizing $E(\beta|X)$. Many times this is done through optimization methods like *gradient descent* [3]. The rationale behind this method can be better understood giving a geometrical interpretation of the minimization problem [22]. Let's consider a simple model with two parameters, β_1 and β_2, and an error function E. The set of possible values for the parameters defines a plane: considering a vertical axis indicating the value of the error corresponding to the values of the parameters, E defines a parabolic surface in the three-dimensional space, with a global minimum. The gradient descent method starts from an initial value of the parameters β_1 and β_2 and then modifies them little by little in such a way that the corresponding value of the error function moves along the surface following the fastest direction towards the global minimum. This direction is represented by the *gradient vector*, whose components correspond to the partial derivatives of E with respect to the respective parameters of the model. Considering a model with p parameters, the gradient vector of the error function E is:

$$\nabla_\beta E = \left[\frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_2} ... \frac{\partial E}{\partial \beta_p}\right]^T$$

Defined as above, the gradient vector actually points to the direction that causes the biggest increase of E, which is why gradient descent takes the negative of the gradient. In particular, the method consists in the application, at each step, for each component i of the parameters vector $\overrightarrow{\beta} = [\beta_1, \beta_2, ..., \beta_p]^T$ of the following rule:

$$\begin{cases} \Delta\beta_i = -\eta \frac{\partial E}{\partial \beta_i} \\ \beta_i = \beta_i + \Delta\beta_i \end{cases}$$

η is positive constant defined as *learning rate* which is needed to limit the size of the updates performed at every step and ensure that the global minimum is not missed. On the other hand, if the rate is too small, the method proceeds very slowly. Having defined the rule for the update of the parameters in terms of partial derivatives of E, it is necessary to also choose the error function in order to get an operative description of the gradient descent algorithm [22]. For example, E can be defined as half of the sum of the squared difference between the actual and the expected output of each input training value. Considering d training input values, being y_i

and $\hat{y}_i$ respectively the expected value and the actual value returned by the model for the i-th training value, E is defined as follows:

$$E(\overrightarrow{\beta}) = \frac{1}{2}\sum_{i=1}^{d}(y_i - \hat{y}_i)^2$$

The partial derivative with respect to the i-th parameter is then:

$$\frac{\partial E}{\partial \beta_j} = \sum_{i=1}^{d}(y_i - \hat{y}_i)(-x_{ji})$$

being x_{ji} the j-th component of the i-th training input, so the corresponding update is:

$$\Delta\beta_j = \eta\sum_{i=1}^{d}(y_i - \hat{y}_i)(x_{ji})$$

In summary, the gradient descent method starts choosing a random vector of parameters and then iteratively updates them applying the update rule, comparing the output corresponding to the current state of the model to the expected output. The method stops when the gradient vector has length equal to zero, namely when the partial derivatives of E are zero: this happens when a minimum of the error function, either local or global, is found.

2.5 Artificial Neural Networks

Artificial Neural Networks (ANNs) draw inspiration from the mechanism employed by the human brain to perform complex operations like vision, speech recognition and learning, to implement these as tasks executable by a computer. The reasons why this approach is promising are to be found in the biological discoveries regarding how the human brain works. It has been indeed shown that its abilities are not due to the computational power of its single "processing units", the *neurons*, but rather on the high number of connections, the *synapses*, between them, which allows an highly-parallel processing. ANNs hence try to emulate this parallelism to improve the performances of the implemented tasks [3, 22].

2.5.1 Perceptrons

Perceptrons are the basic computational units of the ANNs. They take n real input values $x_1, x_2, ..., x_n$ and output a value in the set $\{0, 1\}$. In the simplest case, the result is obtained composing two functions. The output of the first function is a linear combination of the inputs

$$y = \sum_{i=1}^{n} w_i x_i + x_0$$

where each coefficient w_i is a real value referred to as *synaptic weight* (or simply *weight*); w_0 represents a *bias*, so sometimes a "fictitious" input $x_0 = 1$ is introduced in order to express the function as a dot product

$$y = \overrightarrow{w} \cdot \overrightarrow{x}$$

with $\overrightarrow{w} = [w_0, w_1, ..., w_n]$ vector of the weights and $\overrightarrow{x} = [x_0, x_1, ..., x_n]$ vector of the inputs. The perceptron defines a *decision hyperplane*, with equation $\overrightarrow{w} \cdot \overrightarrow{x} = 0$, which divides the $n-$dimensional space of the input values into two halves: on one side $y > 0$, on the other side $y < 0$. That's why usually y is usually used as input of a $signfunction$ defined as follows:

$$sign(y) = \begin{cases} 1, & \text{if } y > 0 \\ -1, & \text{otherwise} \end{cases}$$

In a classification scenario, a set of inputs is said to be *linearly separable* if an hyperplane exists such that each input can be assigned to either side of the hyperplane itself. In a scenario with $m > 2$ output values, there are m parallel perceptrons, each with its own weights vector $\overrightarrow{w_i}$, so the the $i-$th output is:

$$y_i = \sum_{j=1}^{n} w_{ij} x_j + w_{i0}$$

and the output of all the perceptrons as a whole is:

$$y = Wx$$

W is a matrix whose element w_{ij} corresponds to the weight from input x_j to the the output y_i.

2.5.2 Training Perceptrons

A perceptron defines a decision surface which can be exploited for classification, but this requires the perceptron to be trained, namely learning the elements of the weights vector to correctly classify all the given input values. This step can be performed *offline*, before actually using the perceptron, when the whole set of input samples is available. One way of doing it consists in choosing a random initial vector of weights and then adapting it on the basis of the output of the perceptron applied to each input value in the training set. More precisely, for each sample, the weight w_i associated with the input x_i is updated according to the following rule

$$\begin{cases} \Delta w_i = \eta(y - \hat{y})x_i \\ w_i = w_i + \Delta w_i \end{cases}$$

being y and $\hat{y}$ respectively the expected output and the actual output of the percepetron corresponding to the sample; η has the same role as in the gradient descent method. The limit of this method is that it manages to successfully train the perceptron only when the training sample are linearly separable. When this is not the case, the perceptron can be trained applying the gradient descent method, adopting the error function mentioned in 2.4.4. The update rule of the weight applied to the input x_i becomes:

$$\Delta w_i = \eta \sum_{j=1}^{d} (y_j - \hat{y}_j)(x_{ij})$$

y_j and $\hat{y_j}$ are respectively the expected output and the actual output of the percepetron corresponding to the j-th (out of d) sample; x_{ij} is the i-th component of the sample. It has to be said, though, that the weights learned with gradient descent only minimize the error of the *unthresholded perceptron*, namely a perceptron where the sole linear combination of the inputs is applied. In order to adapt the gradient descent to the *thresholded perceptron* (including the sign function), it is possible to train the unthresholded perpeptron with $\{1, -1\}$ as target values. Nevertheless, this does not guarantee that this minimizes the misclassifications of the thresholded perceptron.
When the set of samples is not known in advance, the perceptron is trained with *online learning*: in this case, as new values arrive as input to the percepetron, its weights vector is correspondingly updated to adapt it to the new information available and minimize the misclassification. One common method adopted in this scenario is the *stochastic gradient descent* [3, 22], a variation of the gradient descent method where, instead of iterating over the whole training set before modifying the weight vector, the update is performed after having computed the error associated to a single training example.

2.5.3 Multilayer Perceptrons

Mapping $\{1, -1\}$ to respectively $\{true, false\}$, a simple thresholded function can be used to implement binary functions like logical AND, OR, NAND and NOR. The same does not hold for other boolean fuctions, like the logical XOR though. This is due to the fact that a perceptron defines a linear decision surface, so it can be only used to represent linearly separable functions. When there is the need to define *non-linear* decision surfaces, the solution is represented by *multilayer perceptrons*, with one or more *hidden layers* inserted between the ouputs and the inputs applied to the perceptrons, thus realizing a structure (referred to as *feedforward network*) like the one represented below [3, 22].

The hidden layers are implemented through the *sigmoid unit*. Analogously to the perceptron, the sigmoid corresponds to the application of a function to the linear combination of its inputs, but, unlike the former, in the latter this function coincides with the *logistic function* (or *sigmoid function*) (see Section 2.4.3), defined as:

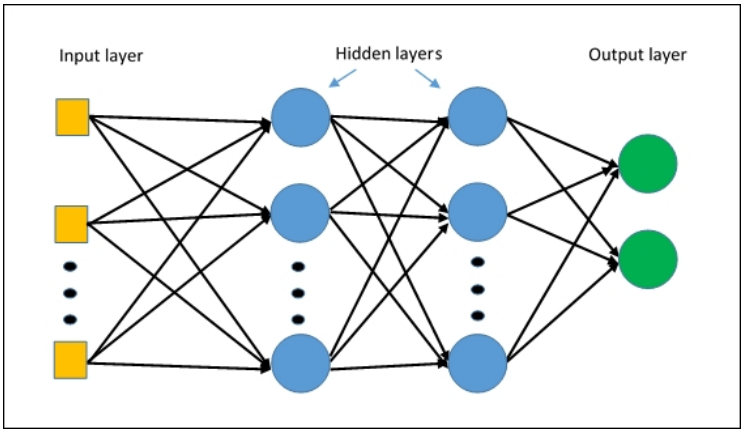

Figure 2.4. Multilayer perceptron

$$\sigma(y) = \frac{1}{1 - e^{y}}, y = \overrightarrow{w} \cdot \overrightarrow{x}$$

The sigmoid function is differentiable, so it can be used in training algorithms based on the gradient descent and also its derivative has an interesting property:

$$\frac{\partial \sigma(y)}{\partial y} = \sigma(y) \cdot (1 - \sigma(y))$$

and its output falls within in $[0, 1]$; an alternative to the sigmoid function is the *tangent function.*

Multilayer perceptrons are trained applying the *backpropagation algorithm*, a method that exploits the gradient descent to minimize the value of the error function corresponding to the squared differences between the expected and the actual outputs of the network. The algorithm iteratively computes the error of the network corresponding to a training sample and modifies the weights of all the units in order to minimize the error itself. The name of the algorithm is due to the fact that the error in the classification committed by each unit of the network propagates downward from the output layers to the bottom hidden layer. Let's consider a network with two hidden layers of sigmoids, each connected to all the units in the preceding layer. In such a network, the update of the weight $w_{i,j}$ from the unit i to the unit j amounts to

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

with $x_{i,j}$ the input from the node i to the node i and δ_j recursively defined as follows [22]:

- for each unit k of the output layer, being y_k and $\hat{y_k}$ respectively the label and the k-th component of output of the network in correspondence of the training sample, $\delta_k = \hat{y_k}(1 - \hat{y_k})(y_k - \hat{y_k})$
- for each hidden unit h with K outputs, being y_h its output in correspondence of the training sample, $\delta_h = \hat{y_h}(1 - \hat{y_h}) \sum_{k=1}^{K} w_{kh}\delta_k$

The rule comes from the application of the *chain rule* of the gradient. Indeed, each weight $w_{i,j}$ only affects the sub-network constituted by the unit j to which it is connected and by all the underlying layers and units in turn connected to it. Referring to this sub-network as $subnet_i$, the derivative of the error function E with respect to $w_{i,j}$ (see Section 2.4.4) is:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial subnet_i} \frac{\partial subnet_i}{\partial w_{i,j}}$$

The same holds for all the weights in $subnet_i$, so the above calculation can be recursively to all the layers of the network, resulting in the aforementioned update rules. Applying the updates to the weights of all the units after having computed the output of a single training example actually provides an approximation of the gradient descent which resembles to the stochastic gradient descent method. In this context, an *epoch* is defined as a single iteration over the training set with the application of the update rules to all the samples.

2.5.4 Variations to the Gradient Descent Algorithm

Vanilla gradient descent implies the calculation of the gradients for all the samples before applying the corresponding updates to the parameters of a model. A training approach applying the vanilla version of the gradient descent algorithm is referred to as *batch learning*, since computation is performed on a large "batch" of data. The problem with batch learning is that it requires to load the entire dataset in memory and this can pose some difficulties if the size of the dataset is big. Moreover, despite gradient descent is guaranteed to converge, this may take a huge number of iterations. To overcome these issues, the vanilla version of gradient descent is usually replaced by stochastic version (see Section 2.5.2), which has been proved to offer nearly the same guarantees to converge, but does this more rapidly; it can be regarded as a particular case of batch learning with batch size equal to one. In practice neural networks are usually trained using a third variant, *minibatch learning*, which falls in between the first two: updates are applied after having taken into account a small portion of the entire dataset; this corresponds to performing batch learning taking batches of size comprised between one and the size of the training set.
Besides varying the number of elements in each batch, there are further modifications to the gradient descent algorithm that help to improve performances in its real applications. One of them consists in altering the update rule for the weights of a neural network by adding a *momentum*. The update applied at epoch n (in case of batch learning) or iteration n (in case of online learning) to the weight $w_{i,j}(n)$, from the unit i to the unit j, becomes:

$$\Delta w_{i,j} = \eta \delta_j x_{i,j} + \Delta w_{i,j}(n-1)$$

being $\Delta w_{i,j}(n-1)$ is the amount of the updated applied to the same weight in the preceding epoch or iteration (sometimes called *momentum term*); is usually takes values from 0.5 to 1. From a geometrical point of view, the added term has the effect of maintaining the direction of the descent along the error function towards the global minimum.

2.5.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialization of ANNs where at least one layer contains an operation called *convolution*. They are particularly suited for computer visions tasks, like image classification or objects detection, because they can handle images of big size better than ANNs. Indeed, since in ANNs a single output in a hidden layer is the linear combination of all the inputs, the number of parameters (weights) grows fast with the size of the input. Images typically contain a high number of pixels, so they are not suited to be used with ANNs. Hence, the typical input to CNNs is an image, usually represented as a third order tensor, where the third dimension corresponds to one channel of the RGB format. The input image is processed though a series of steps, each performed by one layer of the network, which is associate with a set of parameters (weights) analogously to ANNs. In case of image classification, the last layer is often represented by the application of the softmax function (see Section 2.4.3) in order for the network to return a vector of values comprised in [0,1], each indicating the probability that the image belongs to one specific class.
There are three fundamental type of layers:

1. *Convolutional* layer
2. *ReLU* layer, also know as *rectifier*
3. *Pooling* layer
4. *Fully-connected* layer

Convolutional layer

Represents the main component of CNNs and is associated with a number k of kernels, each having a smaller size, f than the input tensor. A kernel can be thought of as a squared filter that gets applied to an image and moved across its width and height, step by step until all the surface of the image has been covered. At each step, the pixels of the image are aligned with those of the filter and the dot product between the two is computed. More concretely, both the image and the filter are vectors and the pixels represent their elements. The output is a *feature map* (or feature), whose elements correspond to the dot products; there is one feature map for each kernel applied to the input. Two parameters regulate the application of a kernel:

1. *stride* s: the number of pixels by which the kernel has to be moved at each step
2. *padding* p: the number of rows of 0s added to pad the input

Being, the input image a third-order tensor with the first two dimensions equal to the width w and height h of the image and c the number of channels (3 for RGB, 1 for gray-scale), the output of the convolutional layer is another third-order tensor with the following sizes for the dimensions:

$$width = \frac{w - f + 2p}{s} + 1$$

$$height = \frac{h - f + 2p}{s} + 1$$

$$depth = k$$

ReLU layer

The rectifier coincides with a simple function that returns the the maximum between 0 and the input. Hence, the result of the composition of the rectifier with another function corresponds to the positive part of the latter. It CNNs it is used to "activate" specific features of an image: applying the rectifier to a feature map, all its negative elements are set to zero, while the positive elements remain unaltered.

Pooling layer

This kind of layer serves to reduce the spatial occupation of the network and the number of parameters, in order to avoid overfitting. The pooling layer divides the input tensor in subregions and assigns to each of them a value that summarizes their content; this can be the maximum or the average of the values in the subregion.

Fully-connected layer

This layer is usually placed as the end of a CNN. As its name suggests, it differs from the other layers in that its output units are connected to all the input units, analogously to hidden layers of ANNs. A fully-connected layer is needed to combine the features learned from preceding layer and making it possible toe learn more complex features in the successive layers. As in ANNs, this layer performs the multiplication between the matrix of weights and the input tensor.

Chapter 3

Integrating Transactional Memories in TensorFlow

The size of datasets available has been increasing much in last few years, both in the commercial and academic fields. This is a good news in the area of machine learning, since it has been shown that training models with a high number of data samples favours the accuracy of the trained model [5, 20]. The amount of data can sometimes be so high that training a model using a single processing unit is not feasible. Given the availability of parallel architectures and distributed solutions, it seems only natural to take advantage of them to speed-up training, which directly translates into parallelizing the application of the stochastic gradient descent (SGD). Several methods have been proposed, but none of them has managed to achieve a definitive solution. This is due to the inherently sequential nature of SGD, which forces to resort to synchronization mechanisms in case of execution on multiple cores or clusters of machines; this, in turn, inevitably limits the performance of training. This holds also for TensorFlow, whose implementation of SGD requires the acquisition of a lock before the application of the updates to the parameters of a model. This dissertation explores the idea of improving the performances of parallel training in TensorFlow replacing the lock-based implementation of SGD with one based on transactional memories: the objective is allowing TensorFlow to speculatively apply SGD without resorting to any synchronization primitive. The next sections provide an overview of the methods available to parallelize the execution of SGD, describe the implementation of this algorithm in TensorFlow and present a new approach based on the integration of transactional memories in the framework.

The problem of training models in parallel using the gradient descent algorithm is well known in literature (see Section 3.1) and trying to solve it by mean of transactional memories represented most of this thesis work. Nevertheless, this is not the only way in which transactional memories can be integrated in TF. In this thesis work, indeed, besides focusing on the elision of a single lock, the completely opposite approach, namely replacing all the locks, was also taken. The last section of this chapter covers this part of the work.

3.1 Parallel Gradient Descent

One of the first works dealing with the parallel execution of SGD, dates back to 2009 [18]. The authors propose a solution where the algorithm is executed by each core of a machine independently and the updates are applied *asynchronously*, i.e. without coordination, to the shared parameters of the model. This can be easily extended also to a distributed setting, with one *parameter server* and a number of *workers*. The former maintains the current state of the parameters of the model, while the latter repeatedly read the values, compute the gradients and send back the related updates. This kind of approach is generally referred to as *asynchronous data parallel training*. The algorithm is a variant of the vanilla gradient descent and is referred to as *delayed gradient descent*: unlike the former, in the latter the update of the parameters at step t does not depend on the gradients computed at time t, but rather on the gradients computed θ steps before. In the aforementioned scenario, being n the number of independent cores that apply SGD on different examples of the training set, the update corresponding to the i-th instance of the data is applied with a delay equal to $n - 1$. Indeed, the results of the computation performed by different cores cannot be applied at the exact same moment, which also implies the need for a read-write lock protecting the current state of the parameters.
Another proposal based on asynchronous computation of SGD was made in 2011 [26]. Here the authors focus on multicore architectures and show that under the assumption of *sparse updates*, namely when SGD modifies at each iteration only a small fraction of the parameters set, it is possible to design a *lock-free parallel* variant of the training algorithm. The absence of synchronization would in general prevent the model from being successfully trained, since the cores would interfere with each other, overwriting the results produced by others. The work, on the contrary, shows that sparsity allows to achieve a speedup of the training that is nearly linear in the number of cores.
DistBelief, the parent of TensorFlow, is intended to offer support for both *model parallelism* and *data parallelism* [6]. Model parallelism consists in partitioning the set of parameters of the model and distributing them across multiple cores in a single machines, or multiple machines in a cluster: the values of the single parameters are learned locally and then the results are merged. Data parallelism, instead, takes place when different instances of the same model are trained at the same time on different cores, or machines, that share the current values of the parameters. DistBelief is based on a modified version of the asynchronous SGD, called *Downpour SGD*, tailored to training large data sets through data parallelism. It requires the data samples to be split into a number of partitions: each of these serves to train independently an instance of the model which is distributed across a cluster of machines representing a *replica*. A replica reads the assigned data samples in mini-batches, sequentially, and for each requests the parameter server for the current state of parameters related to the partition of the model it is in charge of. After having calculated the gradients, the resulting updates are sent back to the shared parameter server. The authors claim that asynchronous SGD achieves a better fault-tolerance with respect to its synchronous version, because of the replication of the model. At the same time, despite it increases the randomness of the learning process, because of the lack of synchronization among workers, and gives fewer

consistency guarantees, it is shown to be effective in practice.

3.2 Gradient Descent in the TensorFlow

3.2.1 Python API

Given the broad usage of the gradient operator in the whole area of machine learning, the developers of TF decided to make its computation automatic, relieving programmers from the burden of taking care of it. The programming model requires users to only build the computational graph (see Section 2.3) representing the error function to be minimized. Once this step has been taken, TF automatically applies the backpropagation algorithm (see Section 2.5.2) to determine the nodes of the graph with respect to which the error function has to be derivated, and connects them with some special `gradients` nodes. These nodes make it possible, when the session is run, to compute the gradient descent in order to minimize the error function and train the corresponding model. Fig. 3.1 [1] shows the computational graph corresponding to the simple snippet of code 3.1 that trains a linear model:

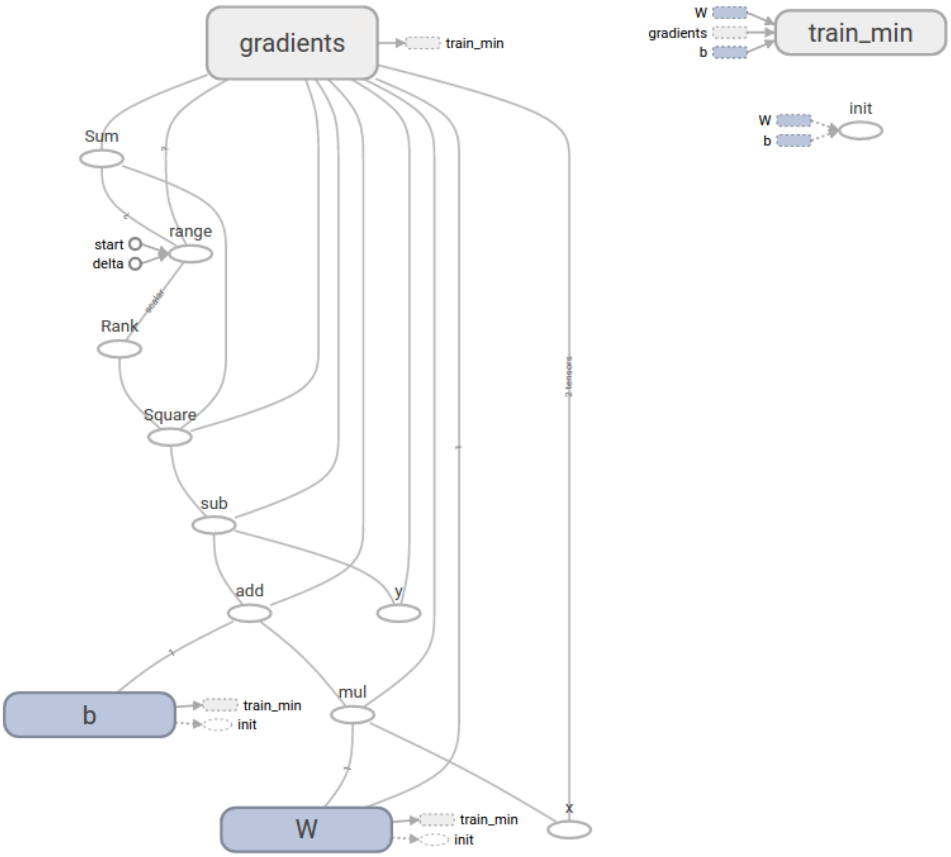

Figure 3.1. The `gradients` in TensorBoard

[1] tensorflow.org/get_started

```
1  W = tf.Variable([.4], dtype=tf.float32)
2  b = tf.Variable([-.5], dtype=tf.float32)
3
4  x = tf.placeholder(tf.float32)
5  model = W*x + b
6  label = tf.placeholder(tf.float32)
7
8  error_function = tf.reduce_sum(tf.square(model - label))
9
10 optimizer = tf.train.GradientDescentOptimizer(0.01)
11 train = optimizer.minimize(error_function)
```

Listing 3.1. Train a linear model

The Python object `GradientDescentOptimizer` is one of the variants of the gradient descent algorithm exposed by the `tftrain` API of TF. In the above listing, invoking its `minimize` method and passing it the handle (i.e. the TF Tensor) corresponding to the error function to be minimized, causes the insertion of the `gradients` nodes in the computational graph. In the above example the object is initiliazed with a `learning_rate` equal to 0.01: this parameter corresponds to the η defined in 2.4.4.

3.2.2 Back-end Implementation

As explained in 2.3.2, each operation appearing in the computational graph of TF is associated with one or more implementations in the C++ back-end, each corresponding to a kernel, namely a specialization of the operation optimized to be run on a particular device. Each kernel is defined as as C++ class, extending the parent class `OpKernel`, whose `Compute` method implements the associated operation. This holds also for `ApplyGradientDescentOp`: this class is in charge of the application of the update rule of the gradient descent algorithm (see Section 2.4.4). In particular, each parameter $\beta_i Section$ in the model is assigned by `ApplyGradientDescentOp` a new value

$$\beta_i = \beta_i + \Delta\beta_i$$

where $\Delta\beta_i$ is $-\eta\frac{\partial E}{\partial \beta_i}$, the derivative of the error function with respect to the parameter, times the learning rate.
As anticipated in 3.1, the execution of this task in a multi-threaded, or distributed, environment requires some form of synchronization, otherwise the updates applied by one thread, or replica, may be overwritten and lost by the updates of other workers. This of course would affect the convergence of the model towards the minimization of the loss function. Nevertheless, TF leaves the programmers the freedom of choosing whether the the code of `ApplyGradientDescentOp` is to be executed inside a critical section guarded by a lock, or not. If more guarantees of consistency are needed, it is sufficient to pass the flag *use_locking=True* when instantiating the object `GradientDescentOptimizer` is instantiated.

3.3 Transactional Application of the Gradient Descent

The main contribution of this thesis work consists in introducing transactional memories in TF in order to avoid grabbing the lock that protects the update of the parameters of a model trained with the gradient descent algorithm. The expectation is that such a modification allows to get a speedup, in terms of execution time, while providing consistency, in the application of the updates. Two different approaches were taken:

1. eliding the lock using Hardware Transactional Memories (HTM)
2. executing the critical section protected by the lock inside a transaction created using Software Transactional Memories (STM)

What follows is a description of the modifications introduced in the back-end of TF in order to enable the usage of these two solutions. They both regarded the piece of code in charge of the execution of the the update rule of the gradient descent algorithm (see Section 3.2.2).

3.3.1 Lock Elision with Hardware Transactional Memory

Given that one of the main causes for transactions to fail is the excessive number of memory accesses (see Section 2.1.4), a convenient preliminary step in the integration of hardware transactional memories consisted in a re-implementation of the code that updates the parameters of a model in `ApplyGradientDescentOp`. Indeed, despite of the simplicity of this task, TF delegates its execution to the Eigen library, in order for it to apply some optimizations under the hood. Unfortunately these optimizations come at the price of a larger memory occupancy, which enlarges the footprint left by the program. As a consequence, in order to make the code more "HTM-friendly", namely reducing the risk of capacity aborts, the original implementation has been simplified, using basic C instructions in place of C++ templates, as in Eigen.

```
1 functor::ApplyGradientDescent<Device, T>()(
2     device, var.flat<T>(), alpha.scalar<T>(), delta.flat<T
      >());
```

Listing 3.2. Original application of the updates, with Eigen

```
1 for(int i=0; i < size_tensor; i++) {
2       var_pointer[i] -= delta_pointer[i] * (*alpha_pointer);
3     }
```

Listing 3.3. HTM-friendly application of the updates, without Eigen

The above code was then wrapped inside a transaction using the interface exposed by the GCC compiler to support transactional memories. As mentioned in 2.2.3, different constructs are to be used depending on the specific architecture where the program is run: since the implementation was tested either on a machine based on Intel TSX and on one based on POWER ISA, two different versions were produced.

Anyway, apart from some syntactic differences between the two, they share the same mechanism, as visible in Listing 3.4 and 3.4

```
1  # define TM_BEGIN(mutex) {
2      while (1) {
3          while (IS_LOCKED(mutex)) {
4              __asm__ ("pause;");
5          }
6          unsigned int status = _xbegin();
7          if (status == _XBEGIN_STARTED) {
8              if (IS_LOCKED(mutex)) {
9                  _xabort(30);
10             }
11             break;
12         }
13         else if (status == _XABORT_CAPACITY) {
14             SPEND_BUDGET(&htm_budget);
15             ...
16         } else {
17             if (status & _XABORT_CONFLICT) {
18                 ...
19             }
20             else if (status & _XABORT_EXPLICIT) {
21                 ...
22             }
23             else if (status & _XABORT_NESTED) {
24                 ...
25             }
26             else if (status & _XABORT_DEBUG) {
27                 ...
28             }
29             else if (status & _XABORT_RETRY) {
30                 ...
31             }
32             else {
33                 ...
34             }
35         }
36         ...
37         if (htm_budget <= 0) {
38             mutex->lock();
39                     break;
40         }
41     }
42 };
43
44
```

```
45 # define TM_END(mutex){
46     if (htm_budget > 0) {
47         _xend();
48         ...
49     } else {
50         mutex->unlock();
51         ...
52     }
53 };
```

Listing 3.4. HTM elision on Intel TSX

```
1  # define TM_BEGIN(mutex) {
2      while (1) {
3          while (IS_LOCKED(mutex)) {
4              __asm volatile ("" : : : "memory");
5          }
6          TM_buff_type TM_buff;
7          unsigned int status = __TM_begin(&TM_buff);
8          if (status == _HTM_TBEGIN_STARTED) {
9              if (IS_LOCKED(mutex)) {
10                 __TM_abort();
11             }
12             break;
13         }
14         else {
15             if(__TM_is_failure_persistent(&TM_buff)){
16                 SPEND_BUDGET(&htm_budget);
17                 ...
18             }
19             else if(__TM_is_conflict(&TM_buff)){
20                 ...
21             }
22             else if (__TM_is_user_abort(&TM_buff)) {
23                 ...
24             }
25             else if(__TM_is_footprint_exceeded(&TM_buff)) {
26                 ...
27             }
28             ...
29         }
30         if (htm_budget <= 0) {
31             mutex->lock();
32             break;
33         }
34     }
35 };
```

```
36
37
38 # define TM_END(mutex){
39     if (htm_budget > 0) {
40         __TM_end;
41         ...
42     } else {
43         mutex->unlock();
44         ...
45     }
46 };
```

Listing 3.5. HTM elision on POWER ISA

The macros `TM_BEGIN` and `TM_END` are placed respectively before and after the code in Listing 3.3 to define the boundaries of the transaction. `TM_BEGIN` is a loop whose code:

1. waits until the lock being elided is free
2. as soon as the the lock is free, tries to start a transaction. On the basis of the outcome of this instruction, the code determines the path to take:
   (a) if positive, it means that the transaction can start, the code exits the loop. But before doing this, it checks if the elided lock is still free and if that's not so the transaction is explicitly aborted because another thread is trying to execute the critical section outside the scope of a transaction
   (b) if negative, it means that the transaction has failed. The value of the dedicated register is read to determine the reason of the failure. In the piece of code used for TF, this check is made only for gathering statistics about the nature of the most frequent failures. Indeed, in any case the transaction is retried, unless the maximum number of attempts has been reached. When this happens, the code exits from the loop and takes the fallback path, namely executes the operation wrapped into the transaction under the protection of the lock

The code in `TM_END` determines whether the transaction has committed or not checking the value of the residual "budget"of the transaction, namely the number of retries left. If the budget is not positive, it means that all the possible attempts to commit the transaction were made, but without success, otherwise it means that the transaction eventually committed. In the former situation, the lock grabbed in the fallback path is released, while in the latter, the the termination of the transaction is requested to the underlying transactional facility.

3.3.2 Application of the Updates with Software Transactional Memory

Despite its recent spread, the hardware support to transactional memories is not so common: that's why in this thesis work also the software implementation of trans-

actional memories was taken into account. The objective was verifying whether it could be advantageous to execute the application of the updates speculatively, inside a transaction, even when TF is run on machines without hardware transactional memories.
TinySTM seemed the most logical choice for introducing STMs in TF, since its C implementation made immediate the integration in the back-end of the framework. TinySTM was combined with the transactional extension of GCC (see Section 2.2.3), which makes it really to take advantage of the transactional support. The programmer is required to only define the limit of the code that has to be run inside a transaction.

```
1 __transaction_atomic {
2     for(int i=0; i < size_tensor; i++) {
3       var_pointer[i] -= delta_pointer[i] * (*alpha_pointer);
4     }
5 }
```

Listing 3.6. GCC TM application of the updates

In Listing 3.6, the instruction `__transaction_atomic` tells GCC that the update of the parameters of the model is to be wrapped into a transaction. At runtime, it is possible, providing the path, to ask for a specific STM back-end, for example TinySTM, to run the transaction; otherwise, the default *GCC TM Runtime* is invoked.

3.4 Eliding All the Locks in TensorFlow

While it was necessary to modify the source code of TF to elide the lock that guarded the application of the updates, it was not feasible to apply the same approach to elide all the locks grabbed during the execution of a Python program based on TF. As a consequence, it was necessary to apply the code in charge of the elision transparently to the back-end of TF. The solution consisted in changing the internal implementation of the locking mechanism adopted in TF. In this way, every time the program needed to enter in a critical section and requested to acquire a lock, the critical section was initially executed speculatively as a transaction, without actually grabbing the lock. If the transaction managed to commit, the lock had been successfully bypassed and the effects of the operations in the critical section were made visible. Otherwise, if the transactions failed for a number of times without committing, the critical section was effectively run with the protection of the lock.
Since the back-end of TF is written in C++, the underlying implementation of the locks is provided by the class `std::mutex` of the standard library, and this in turn relies on the *pthread (POSIX thread)*, which is part of the *glibc* library. Thanks to the work by Andi Kleen [17], it is been now for about five years that pthread offers the support to lock elision. In particular a new "elision path" has been added to the functions that implement the synchronization primitives offered by pthread. Such a path is a wrapper placed around the lock and tries to elide it through a transaction;

if the transaction can't commit, the lock is acquired as a fallback. The transactional execution of the critical section is based on the RTM interface of Intel TSX (see Section 2.2.1), so the code in the elision path is similar to the one in Listing 3.4, except for what concerns the management of failures of a transaction. An *adaptive elision algorithm* is used in order to avoid that a program gets slowed down because of a transaction that never commits despite of many retries (which cause a waste of time). In case of abort, the algorithm, rather than immediately re-running the transaction associated with a lock, may temporarily disables the elision for that specific lock for a certain period; as soon as this period has elapsed, the elision path is reactivated. On the basis of the reason that caused the transaction to abort, the algorithm decides whether the elision should be suspended and, in this case, the duration of the suspension. In particular, four parameters are used to regulate the behaviour of the algorithm:

1. `skip_lock_busy`: if the transaction was aborted because the lock was busy, the value of this parameter defines the number of successive acquisitions of the lock that will avoid to take the elision path

2. `skip_lock_internal_abort`: if the transaction was aborted for reasons other than the conflict with another thread, the value of this parameter defines the number of successive acquisitions of the lock that will avoid to take the elision path

3. `skip_trylock_internal_abort`: same as `skip_lock_internal_abort`, but for try locks

4. `retry_try_xbegin`: if the transaction was aborted but there is still a chance for a commit, the value of this parameter defines the number of times a transaction can be re-attempted

By default the elision path is not active in glibc: it can be enabled re-compiling the library with the flag `-enable-lock-elision=yes` and this causes the elision to be automatically tempted by default on every lock. In alternative, thanks to a tool by Andi Kleen [2], it is possible to activate the lock elision and tune the aforementioned parameters at runtime, setting the `LD_PRELOAD` environment variable on Linux. This is how, as described in the second part of Chapter 4, the transactional elision could be applied to all the locks of TF.

[2] github.com/andikleen/tsx-tools/tree/master/glibc-tune

Chapter 4

Experiments and Results

In this chapter are presented the results of two kinds of experiments conducted to verify the effects of the integration of the transactional memory support in TensorFlow. The first group of experiments was performed to verify whether the application of the updates to the parameters of a model in a transactional manner has some positive effects with respect to the original implementation, when this is protected by a lock. Positive effects are to be intended here in terms of speedup, namely reduction of the time necessary to train the model, against the accuracy of the trained model. The expectation was that transactional memories, thanks to their speculative execution, allowed to remove, in a context of parallel computation, the bottleneck created by the acquisition a lock before updating the parameters. At the same time, transactional memories were supposed to guarantee that, despite of the high degree of parallelism in the application of the updates, this operation was done in a way that was equivalent to the case where workers were executed sequentially. More precisely, the underlying implementation of the transactional memory was intended to avoid that workers could overwrite other worker's progresses. This, otherwise, would have required more iterations of the training to reach the same level of accuracy with guaranteed by the lock-protected scenario. The tests were performed taking into account both hardware (HTM) and software (STM) implementation of the transactional memories. As regards with HTM, the tests were run on two machines with different hardware transactional support:

1. one featuring two Intel(R) Xeon(R) E5-2648L v4 [1] CPUs, based on Intel TSX (see Section 2.2.1), with the support to up to 64 simultaneously running threads (having enabled HyperThreading)

2. a POWER 8 machine [2], based on POWER ISA (see Section 2.1.5), with the support to up to 80 simultaneously running threads

A second set of experiments was aimed at checking if the whole TF library can profit from the usage of transactional memories. This series of experiments can be regarded as a super-set of the preceding one, since the Transactional Lock Elision was tempted on all the locks existing in the code of the framework, not just the

[1] ark.intel.com/it/products/61426/Intel-Xeon-Processor-E5-2648L-20M-1_80-GHz-8_0-GTs-Intel-QPI

[2] www-03.ibm.com/systems/it/power/hardware/s822

ones related to the gradient descent algorithm. It was possible to run this second group of experiments only on the first of the aforementioned machines, because of the absence of the automatic lock elision support by glibc on POWER ISA (see Section 3.4).
The various versions of the framework, obtained with the introduction of the support to transactional memories, were tested with two Python programs, each implementing the training the model by mean of the API of TF. The models are quite different in terms of algorithmic complexity, but they are trained using the same approach and on the same dataset. They represent two different approaches to the same learning problem: recognizing decimal digits from their handwritten depictions.
The next section gives a description of the dataset and of the training approach. An overview of the two models follows and, last but not least, are reported the results of the experiments.

4.1 The MNIST dataset

This dataset is intended to be used to train models for performing recognition of handwritten images representing decimal digits. It contains 28x28 binary images of handwritten digits [21], split into two subsets: a training set with 60000 images and a test set with 10000 images; the digits were written by approximately 250 different people. MNIST originates from two other datasets (NIST's Special Database 3 and NIST's Special Database 1) which contain binary images of handwritten digits; the two halves of elements in MNIST come respectively from the two datasets. The Python API of TF features a dedicated method to automatically load MNIST into memory.

```
1 from tensorflow.examples.tutorials.mnist import input_data
2
3 mnist = input_data.read_data_sets("MNIST_data/", one_hot=
    True)
```

Listing 4.1. Load MNIST in Python

The `mnist` object three subsets from MNIST:

- `mnist.train` contains 55000 train images and corresponding labels
- `mnist.test` contains 10000 test images and corresponding labels
- `mnist.train` contains 5000 validation images and corresponding labels

Images are stored as vectors whose 28x28=784 elements are equal to the intensity of its pixels (a value between 0 and 1); the label corresponding to each digit i is represented by a vector with ten elements: they are all set to 0 apart from the element at index i, which is insted equal to 1.

4.2 Training of the Models

In order to minimize the loss function, the gradient descent algorithm is applied with the learning rate recommended for this dataset, equal to 0.5. With respect to the

version of the program presented by the official TF documentation, the one used for the experiments here described was slightly modified in order to enforce the adoption of an *asynchronous data parallelism* [2]. In particular, after the main process has created a TF Session, with its associated computational graph representing the model to be trained, it launches a number of Python threads with the role of workers (see Section 3.1. Each of them reads the MNIST dataset in mini-batches, sequentially, computes the gradients of the loss function and applies them to the parameters of the linear model (β_0 and β_1). There is no coordination among the workers, so they do not wait each other before applying the results of their computation: this determines an asynchronous scenario. Moreover, the actual state of the model is shared and guarded by a lock: indeed, the gradient descent is run with the *use_locking* flag set to `True`, so the application of the updates is serialized.
One point to be remarked regards the size of each batch read by a worker at each iteration. Being *batch_size* the number of input images in a batch, considering that each image is a vector of 784 `float` values, the memory occupancy of the batch is:

*batch_size**784*`sizeof(float)`

Considering the limited capacity of the cache memories used to store the memory accesses of a transaction (see Section 2.1.6 and 2.1.5), the risk is high that transactions fail because such a limit is overcome. In order to mitigate this risk, a trick adopted in some of the experiments next described, consisted in truncating a portion of the bytes of the input image before they were used for the training. The number of images after this cut corresponds to the `Matrix` parameter appearing in the plots of the experiments.
The last note regards the number of iterations performed over the dataset: they are split equally among all the workers, so the `Iterations` parameter in the graphs later shown refers to the sum of the iterations done by the workers.

4.3 Evaluation of the Models

As said, the accuracy of the model is one of the metrics used to evaluate the performances of the trasactified version of TF. For the decimal digit i, the label y is a vector with ten zeros, except the for the i-th element, which is set to 1.

$$y = [0, ..1, ..0]$$

The output of the model, when it is given in input a representation of the same digit is

$$y = [y'_0, y'_1, ...y'_9]$$

being $y'_i \in [0, 1]$ the estimated probability of the input image of representing the digit i.
If the prediction is correct, the highest element of y'_i, namely the most probable digit, is exactly i. Considering *argmax*, a function that returns the index of the highest element in a vector, the model is correct if

$$argmax(y) = argmax(y')$$

Hence the accuracy of the model is calculated as the fraction of input samples for which the above equation holds.

4.4 MNIST Softmax Regression (`mnist_softmax`)

The first model derives from one of the tutorials present in the official site of TF[3]; in the rest of the dissertation we will refer to it as `mnist_softmax`. Recognizing digits from their handwritten representation is one of the most common examples of *pattern recognition* [3]: the main difficulty is caused by the fact that pictures in the training set usually come from different people, each having his own handwriting style. The task can be considered a classification problem where each handwritten image has to be assigned one out of ten classes, each corresponding to a decimal digit. This problem can be tackled with a supervised learning approach. The model learns to calculate an estimation of the probability that an image belongs to a class, where each class is associated with a decimal digit. For an input image, such an estimation is computed for each class and then the digit corresponding to the highest probability is returned. This probability can be evaluated with a multiclass logistic regression model (see Section 2.4.3), where:

- the variable X coincides with the vector describing the pixels of an input image
- the variable Y is a column vector with ten elements where its i- th component amounts to the probability of the input image of being the handwritten representation of the i-th digit
- the parameter β_1 that is to be learned is a matrix with 784 rows (as many as the number of pixels in each image) and 10 columns (as many as the binary digit)
- the parameter β_0 (*bias*) to be learned is a column vector with ten elements

The component of the output related to the each digit i can be computed applying the softmax function as follows:

$$Y_i = \frac{e^{\beta_{1i}} X + \beta_{0i}}{\sum_{j=0}^{9} e^{\beta_{1j} X + \beta_{0j}}}$$

being β_{1i} the i-th row of the matrix β_1 and β_{0i} the i-th element of the bias. As already mentioned in 2.4.3, a proper choice for the error function to be minimized is the cross-entropy, which in this case, for each image of the traing set is equivalent to:

[3] www.tensorflow.org/get_started/mnist/beginners

$$E = -\sum_{i=0}^{9} y_i \log(\hat{(}y_i))$$

where y_i is the i-th component of the label of the image and $\hat{p}(x_i)$ is the estimated probability that the image represents the digit i. The parameters of the model, β_0 and β_1 are learned using the stochastic gradient descent (see Section 2.4.4) to minimize the cross-entropy.

4.5 MNIST Convolutional Network(mnist_deep)

Also the second model is proposed in one of the tutorials available in the official site of TF[4]; in the rest of the dissertation we will refer to it as `mnist_deep`. It is based on a convolutional network (see Section 2.5.5) and represents a more sophisticated approach to the same task as in `mnist_softmax`, namely the recognition of handwritten digits. Being more complex, `mnist_deep` is theoretically capable of achieving on average a greater accuracy than its linear counterpart (`mnist_softmax`). The network is fed with the images from the MNIST dataset, converted to four-dimensional tensors with one element set 1 to indicate the number of input channels (the images are gray-scale); the output is the same as in `mnist_softmax`. The network is composed of the following layers:

1. the first one combines a convolution with a max pool function. Each kernel of the convolutional layer has a size of 5x5 and maps each image to 32 features. The result of the convolution is added to a vector of small biases, before passing through a rectifier. The biases serves to guarantee that all the features are activated. The last stage of the layer corresponds to the max pool function, which shrinks tensors reducing their size to 14x14

2. the second layer has the same structure as the first one, but has 64 output features; the max pool function further reduces the size of output tensor to 7x7

3. the third layer is fully-connected: it receives in input a 7x7x64 tensor and connects each of its element to 1024 outputs, thus combining the information learned from the single features. Also in this case a bias is added to the result of the matrix multiplication and the sum is activated by a rectifier

4. the last layer has the same role as the softmax function in the previous model: producing a vector of probabilities, one for each digit. To obtain this vector, the output from the previous layer is multiplied by a weight vector. The risk of overfitting data is mitigated by the application of a *dropout*, which sets to zero the weights according to probability function provided in input

Like in `mnist_softmax`, the cross entropy is the loss function to be minimized and the gradient descent algorithm is used to tune the parameters of the layers.

[4] www.tensorflow.org/get_started/mnist/pros

4.6 TLE of the Lock for the Application of the Updates

This section contains the results regarding the first set of experiments, where the lock that protects the application of the updates derived from gradient descent algorithm is elided using both the hardware and the software transactional support. Results are presented through two kinds of plots: *speedup plots* and *accuracy plots*.
The speedup plots compare the improvement, in terms of reduction of the training time, deriving from the parallel execution of the gradient descent (see Section 3.1) when different configurations of the algorithm are adopted. The training time measures the time required (in seconds) by a Python program to train a model using the gradient descent, in TF. Being $t_{parallel}(w)$ the training time corresponding to the parallel execution of the algorithm with a number of w workers (defined `Training Threads` in the plots), and t_1 the training time for the same model when a single worker is used, a line in the speedup plots represents the function:

$$speedup(w) = \frac{t_1}{t_{parallel}(w)}$$

Each line is associated with one of the following configurations:

- `HTM`: the update of the parameters of the model is protected by a lock (*use_locking=True*); the lock is elided with the hardware transactional support
- `STM`: the update of the parameters of the model is protected by a lock (*use_locking=True*); the lock is elided with the software transactional support
- `Lock`: the update of the parameters of the model is protected by a lock (*use_locking=True*)
- `NoLock`: the update of the parameters of the model is **not** protected by a lock (*use_locking=False*)

Being $a_{parallel}(w)$ the accuracy, in the recognition of handwritten digits, achieved by a model trained in parallel with w workers, each line in the accuracy plots shows the values of $a_{parallel}(w)$ in the configurations mentioned above.

4.6.1 Elision with HTM

The graphs contained in this section refer to the experiments in which the lock protecting the application of the updates was elided leveraging the transactional memory facility of Intel Haswell and POWER8.

Additional Plots: Commits and Failures

Since the implementation of the hardware support to transactional memory offered by both the architectures is best-effort (see Section 2.1.6 and 2.1.5), transactions can fail. On both architectures (see Section 4.6), in case of failure the adopted strategy consists in trying to retry the transaction for a number of times; if none

of the attempts succeeds, the lock is acquired to perform the critical section in a non-transactional way. This series of experiments keeps track of the number of times that a transaction managed to commit: this number is referred to as *tle commits*. Dedicated graphs show, as the number of workers increases, the variation in the percentage of tle commits. More formally, being $tle(w)$ the number of tle commits obtained with w workers, and $transactions(w)$ the total number of transactions performed by these workers, the function plotted is:

$$commits(w) = \frac{tle(w)}{transactions(w)} * 100$$

Other plots are dedicated to the reasons behind the failure of the transactions. Such motivations are identified reading the values of the `EAX` register in Intel TSX and of the `TEXASR` register in POWER ISA, thanks to the macros exposed by the GCC interface (see Section 2.2.3).
On POWER ISA, failures of transactions are grouped in six classes:

1. *capacity abort*: the transaction was aborted because the size of memory accessed didn't fit the cache of the processor; the failure is identified with the `__TM_is_footprint_exceeded` macro
2. *user abort*: the transaction was aborted because the user aborted it; the failure is identified with the `__TM_is_user_abort` macro
3. *self conflict*: the transaction was aborted because it performed an instruction not allowed (like trying to nest a transaction beyound the maximum of the nesting level); the failure is identified with the `_TEXASR_SELF_INDUCED_CONFLICT` macro
4. *transactional conflict*: the transaction was aborted because it conflicted with another transaction; the failure is identified with the `_TEXASR_TRANSACTIONAL_CONFLICT` macro
5. *non-transactional conflict*: the transaction was aborted because it conflicted with another thread running outside the scope of a transaction; the failure is identified with the `_TEXASR_NON_TRANSACTIONAL_CONFLICT` macro
6. *other abort*: the transaction was aborted because of a cause that does not fit any of the above definitions

The related plots show, as the number of workers increases, the variation of the percentage of failures and of the relative percentage of each kind of failure. The plots consist in stacked bars, with each bar divided in six sub-bars, one for type of failure. The height of each sub-bar is computed as follows. For each class of failures x, being:

- $x(w)$ the number of transactions that failed because of this kind of failure (training the model with w workers)
- $tle(w)$ the number of tle commits obtained with the same workers

- $failures(w)$ the total number of failures obtained with the same workers

the height of the sub-bar for x is:

$$fail_x(w) = \frac{x(w)}{failures(w) + tle(w)} * 100$$

Note that considering the number of tle commits in the denominator gives an immediate visual feedback about the ratio between failures and commits: given a number of workers, the higher the stacked bars, the higher the number of failures with respect to commits, and vice versa.
On Intel TSX the identification of the reason of conflicts is coarser, so only three kind of failures are considered:

1. *capacity abort*: the transaction was aborted because the size of memory accessed didn't fit the cache of the processor
2. *conflict*: the transaction was aborted because of a conflict
3. *other*: the transaction was aborted because of a cause that does not fit any of the above definitions

Failures are plotted with the same stacked bars as in POWER 8, but with only three sub-bars.

Experiments with `mnist_softmax`

The first plot (Fig. 4.1) shows the speedup achieved on Intel TSX, using batches of 100 images and performing a total number of iterations over the MNIST dataset equal to 1000; the images were not truncated (see Section 4.2), since the value of the input images used, indicated by `Matrix` coincides with their original size.
It is immediately visible that with this configuration of parameters:

1. there is no sensible difference between the case when the gradients are applied holding the lock and the case where this operation is wrapped into transactions
2. there is no sensible difference between the case where the updates are applied with some guarantees of consistency (provided by either locks or transactions)
3. the model does not scale well: while it is almost perfectly linear up to 16 training threads (workers), after it gets sublinear

As regards with the first observation, a possible explanation is that the lock being elided, that protects the update of the parameters of the model, is not *contended*. This means that the as soon as a worker tries to acquire a lock, it manages to do that, without having to wait, because no other worker is simultaneously holding it. To justify the phenomenon highlighted in the second observation, it is useful to check the "behaviour" of the transactional facility in terms of commits and conflicts. As for commits (Fig. 4.2), it can be noted that the percentage of tle commits is always below 50%, independently on the number of training threads. From Fig. 4.3

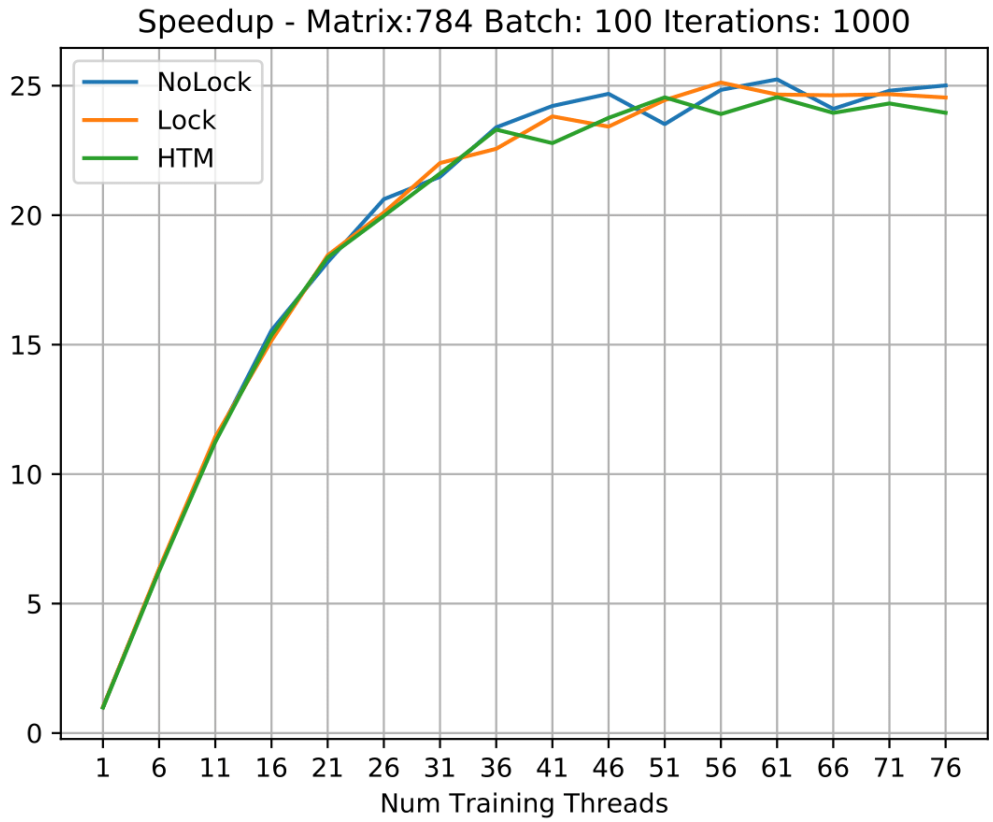

Figure 4.1. Speedup with HTM elision on Intel TSX - `mnist_softmax` - size:784

it can be seen that the main reasons for failures are capacity aborts and conflicts. The fact that most of the times it is not possible to commit the transaction explains why the `HTM` configuration matches the `Lock` configuration: the necessity of often making new attempts to commit the transaction before giving up and taking the fallback path, causes a loss of time which compensates the few cases where the transaction can commit.

A further attempted was made, maintaining this configuration but truncating the images to size of 100. The rationale behind this slightly different attempt was that the reduction of the footprint of the transaction was expected to reduce the number of capacity aborts.

The plots in Fig. 4.4 clearly show a reduction in the number of capacity aborts, with all the number of workers.

A further proof that the transactional memory support benefits from the shortening of input images, derives from the plots relative to the percentage of tle commits.

In Fig. 4.5 the percentage of commits stays over 50% at least up to a number of training threads equal to 26; after this threshold, the trend is similar to the case with the bigger matrix. Given these premises, is was reasonable to expect a wider difference in the behaviour between the `Lock` and `HTM` configurations with respect to the case with `Matrix` equal to 784.

In reality, as witnessed by Fig. 4.6, the transactional application of the updates

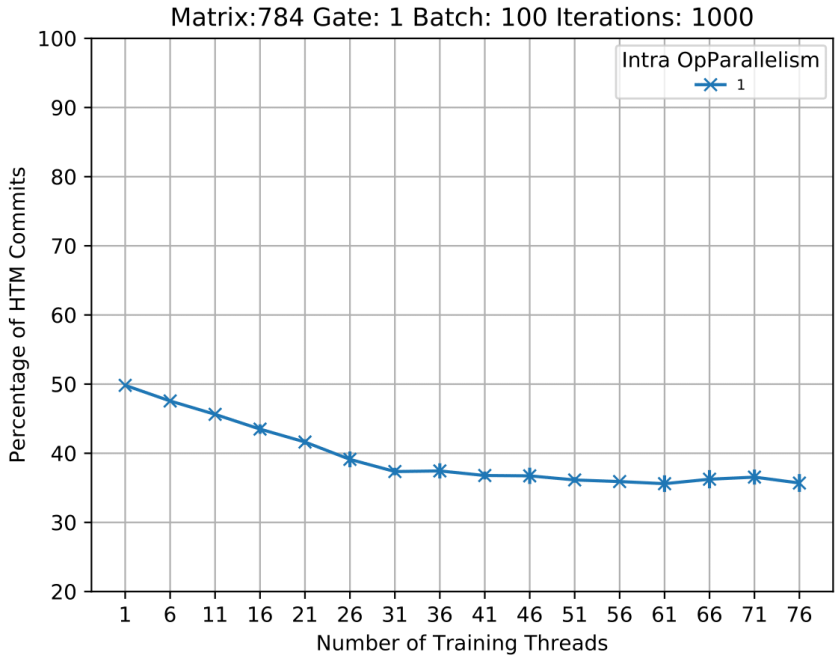

Figure 4.2. Commits with HTM elision on Intel TSX - `mnist_softmax` - size:784

is equivalent to the case where a lock is grabbed. Another thing that emerges is that decreasing the size of input images reduces by much the maximum speedup achievable: with `Matrix=784` it is around 25, while here it is around 8.
Given the not so-promising results obtained on Intel TSX, the same model was trained also on POWER8. Despite trying multiple combinations for the size of images, size of the batches and number of iterations, the only one providing some positive results was: `Matrix=784`, `Batch=1`, `Iterations=10000`.
There are two really encouraging results in Fig. 4.7:

1. the plot shows a clear difference between the configurations `Lock` and `HTM`
2. the transactional supports outperforms the synchronization scheme based on "lock"

The fact that here it is clearly more efficient, in terms of time, to update the parameters of the models inside transactions than in lock-protected critical sections can be explained considering the minimal size of each batch. Indeed, this ensures a high frequency in the application of the gradient descent (it is applied after reading a single image from MNIST) and consequently causes the advantage of applying the updates speculatively to emerge. This is due to the fact that, with this particular configuration, the percentage of time required for the application of the updates with respect to the total time necessary for the whole training phase is high.
Looking at the plot in Fig. 4.8, it is possible to verify that also on POWER8, like on Intel Haswell, using the input images without truncating them causes the limit of the cache to be exceeded most of the times, resulting in a high number of capacity aborts. This is not surprisingly, considering that the limit for the footprint of the

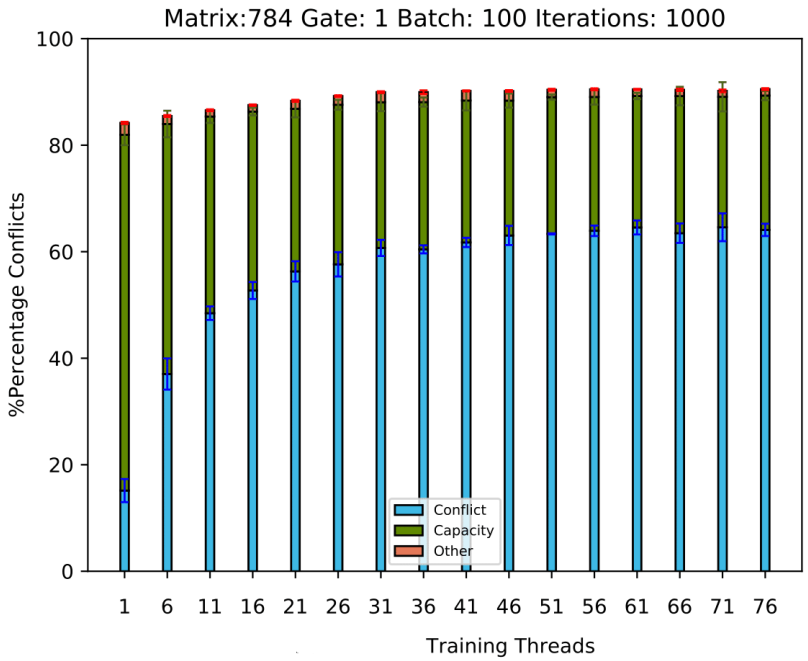

Figure 4.3. Conflicts with HTM elision on Intel TSX - `mnist_softmax` - size:784

transaction is higher in Intel Haswell than in POWER8.
It does not come as a surprise the fact that the configuration without the acquisition of the lock (`NoLock`) is faster than the one that uses transactional memories, because no time is spent for the coordination of the workers.

In the graphs of Fig. 4.9 it can be observed that the accuracy of the trained model achieved in the `NoLock` and `Lock` scenarios essentially coincides. An obvious conclusions is that in the simple linear model trained in `mnist_softmax` it is actually useless to protect the application of the updates with a lock. This is probably due to the lack of contention on the lock observed in the experiments run on Intel. This is confirmed by the low percentage of conflicts on POWER 8 (Fig. 4.8).
In summary, it can be said that the introduction of the hardware transactional support to train the linear model of `mnist_softmax` is worthy only on POWER ISA, using mini-batches, and only in those scenarios where the protection of the application of the updates with a lock is mandatory.

Experiments with `mnist_deep`

Various combinations of parameters were tempted also in the set of experiments regarding the training of the convolutional network. Nevertheless, the only parameter whose variation really brought to heterogeneous results was, again, the batch size.
From the plot in Fig. 4.10 it is possible to evince that the scalability of the model has a behaviour similar to the one observed for `mnist_softmax`: as the number of workers increases, the trend is first linear, up to 15 training threads, than sublinear and finally tends to stabilize, despite of some spikes (with 34 and 55 workers). More

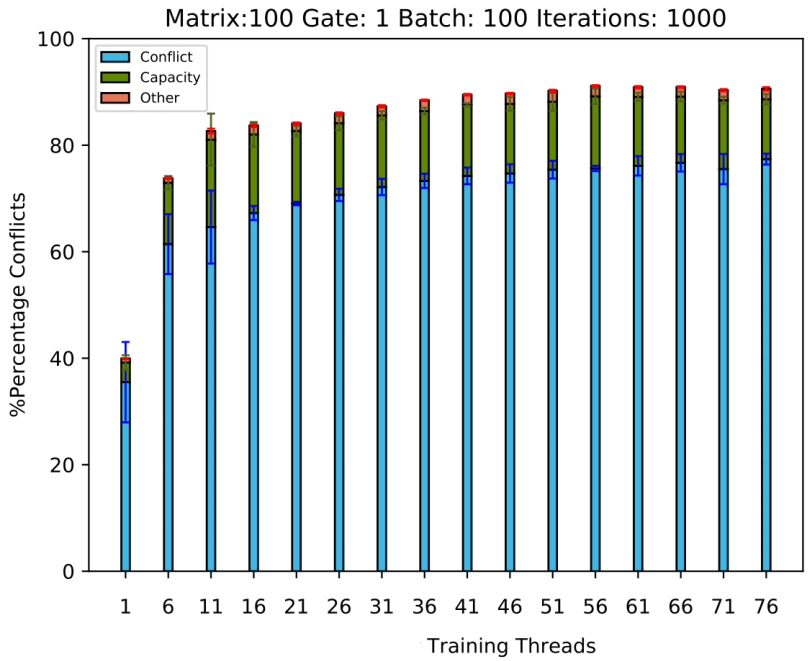

Figure 4.4. Conflicts with HTM elision on Intel TSX - `mnist_softmax` - size:100

importantly, it can be seen that the usage of the transactional support is beneficial from the point of view of the execution time. Indeed, as expected, the line indicating the speedup achieved with the `HTM` configuration lies in between the lines of `Lock` and `NoLock`. As anticipated, this no longer holds as soon as the batch size is slightly increased, as witnessed by Fig. 4.11. Despite the batch size is still very small (only 15 images), the fact that the lines for the three configurations almost perfectly coincide suggests that the phase of computation of the updates takes far longer time than applying them. This makes almost irrelevant with respect to the global training time, whether some time is spent grabbing a lock or not. This implies that also for this model, it is necessary to focus only on the scenario with mini-batches.
In the setting, with a number of iterations equal to 100, the accuracy is really low, independently of the configuration used, and this tells that the complexity of the model requires to go many more times over the dataset. Nevertheless, the graph in Fig. 4.12 shows that even with a high number of iterations (5000) the achieved accuracy is still not acceptable.

4.6.2 Elision with STM

The graphs contained in this section refer to the experiments in which the application of the updates derived from the execution of the gradient descent was performed inside a transaction with a software support. The results obtained from the application of the hardware transactional memories to `mnist_softmax` suggested the trained model was not complex enough to highlight the differences between a scenario where the parameters are updated holding a lock and one where they are directly updated without synchronization. That's why the attempts to integrate the software

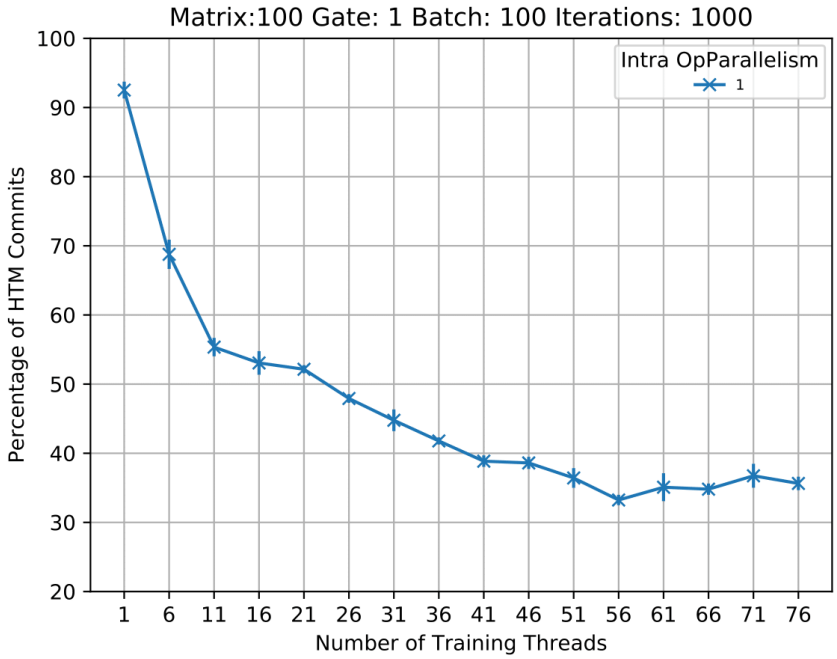

Figure 4.5. Commits with HTM elision on Intel TSX - `mnist_softmax` - size:100

transactional support in TF took into account just `mnist_deep`. Another precious information learned from the preceding set of experiments regarded the size of the batches used to train the models: it was worthy to focus on the scenario with mini-batches.
As anticipated, TinySTM was the chosen software transactional system chosen for the experiments next described. It gives programmers the opportunity of tuning some parameters of the implementation at compile time. Among these, one of the most relevant is the one that sets the behaviour of the system in case of conflict. Two different settings were used:

1. `CM_SUICIDE`: corresponding to an "eager" strategy where a transaction is aborted as soon as a conflict is detected

2. `CM_BACKOFF`: also in this case the transaction is aborted in case of conflict, but then, after some *backoff time*, the transaction is restarted again. The delay between two successive retries is randomly chosen in a range of values that grows exponentially with the number of attempts

The plots refer to results obtained running a benchmarker on Intel Haswell: they present a comparison among the software transactional support (`STM`), its hardware counterpart (`HTM`); the `No Lock` `Lock` are also reported as a reference to the original implementation of TF.

Experiments with Default Contention Management

The plots in Fig. 4.13 and Fig. 4.14 clearly show that the speedup of the training phase achievable using the software implementation of the transactional memories

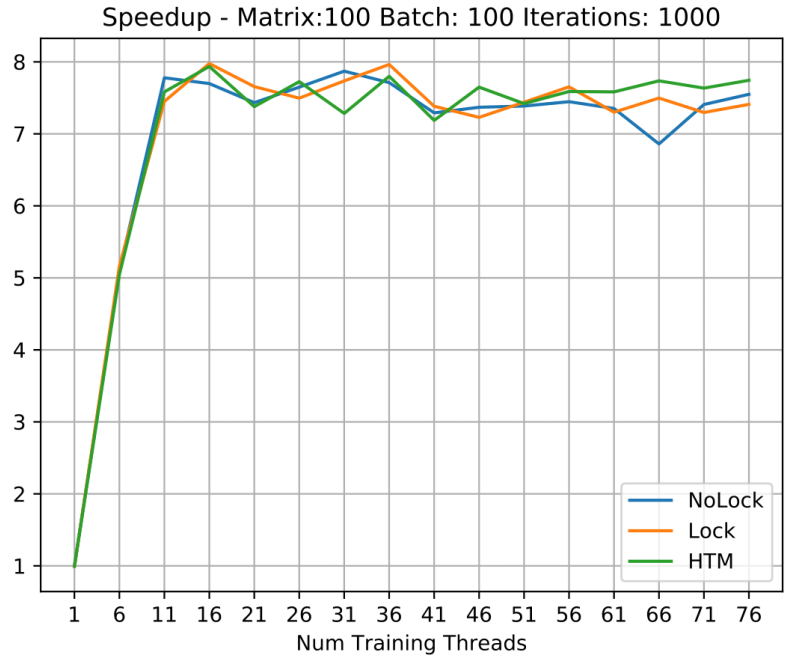

Figure 4.6. Speedup with HTM elision on Intel TSX - `mnist_softmax` - size:100

is very poor, independently of the number of workers. The graphs related to the accuracy of the model trained with the STM support are not shown since its application is proved to be very inefficient in this configuraion.

Experiments with Backoff Contention Management

The plots in Fig. 4.15 and Fig. 4.16 almost perfectly match those obtained with the other strategy for the contention management. This means that retrying to execute the transactions with some delay between successive attempts does not affect the performances of TinySTM, in this configuration. It is pointless to report the graphs related to the accuracy, for the same reason as before.

4.7 Global TLE

The last sequence of experiments regarded the transactional elision of all the locks present in the framework and it was achieved activating the relative support in glibc library. Again, the chosen scenario was the training of the convolutional network (`mnist_deep`) using mini batches.
Two different combinations of parameters of the adaptive elision algorithm were tried. The first one, with the following values:

1. `skip_lock_busy=3`
2. `skip_lock_internal_abort=3`
3. `skip_trylock_internal_abort=3`

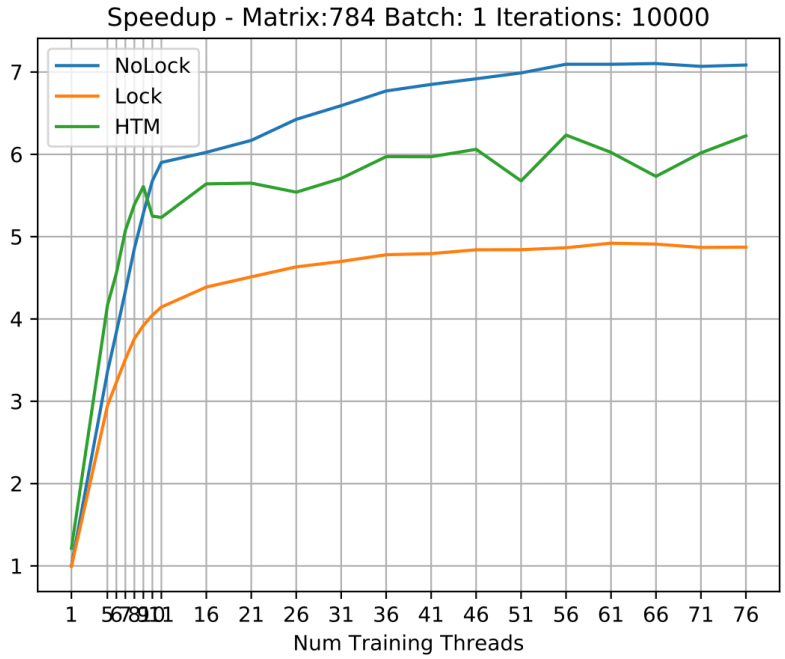

Figure 4.7. Speedup with HTM elision on POWER ISA - `mnist_softmax` - size:784

4. `retry_try_xbegin=3`

This corresponds to the default configuration of the algorithm, where the lock elision is suspended for the next three acquisitions in case a transaction fails without any hope that it can eventually commit. The other combination of parameters

1. `skip_lock_busy=0`
2. `skip_lock_internal_abort=0`
3. `skip_trylock_internal_abort=0`
4. `retry_try_xbegin=5`

was radically different. Indeed, setting the first three values to zero causes the lock elision to be tempted on all the locks, even when the it seems pretty clear that the transaction will never commit. Also, the limit for the maximum number of attempts was brought from 3 to 5, in order to give a transaction more chances to commit. Given the substantial differences in the setting of the elision algorithm, the expectation was to observe some differences in the relative results, but Fig. 4.17 and Fig. 4.18 show no. Is to be noted also that in both cases the speedup is the same obtained with the HTM configuration. One possible explanation for this is that none of the locks in TF can be elided except for the one that protects the application of the updates. Indeed, the critical section protected by the latter is very short, and its memory occupancy was reduced bypassing the Eigen library. On the contrary, for the rest of the locks, the critical section is more complex and, as consequence, not "HTM-friendly', causing a high percentage of capacity aborts.

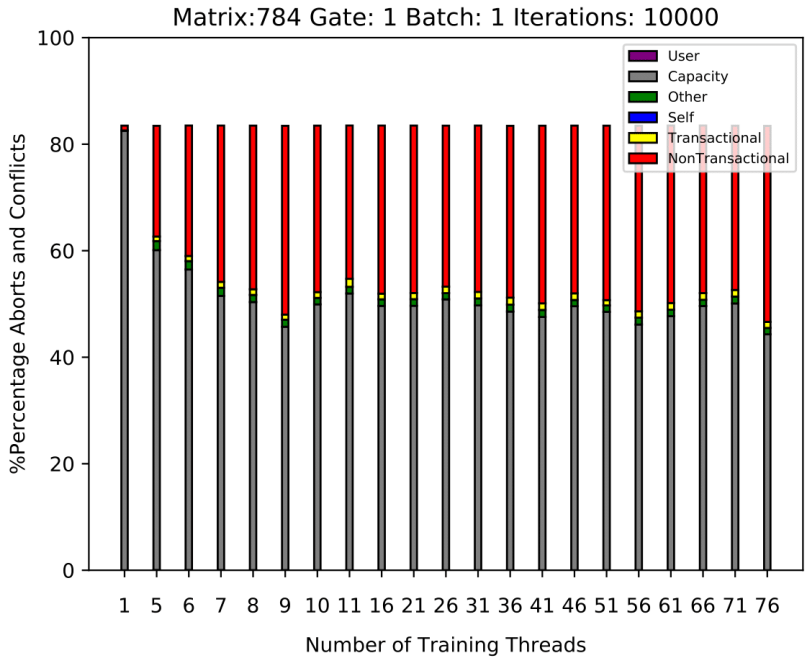

Figure 4.8. Failures with HTM elision on POWER ISA - `mnist_softmax` - size:784

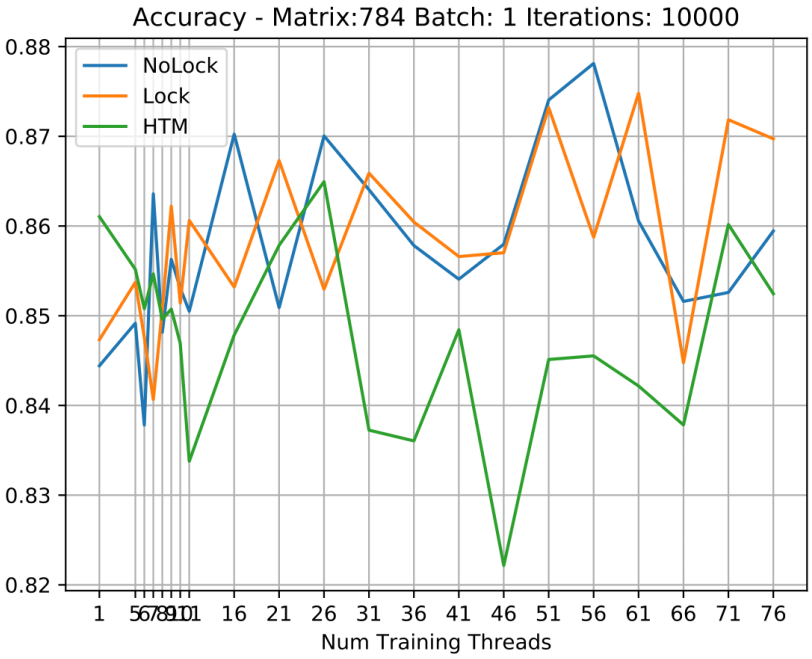

Figure 4.9. Accuracy with HTM elision on POWER ISA - `mnist_softmax` - size:784

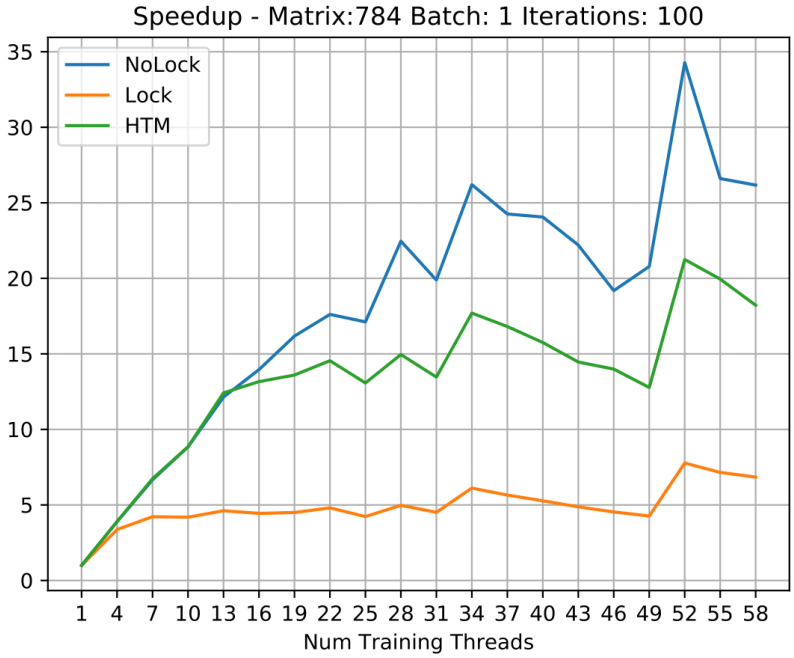

Figure 4.10. Speedup with HTM elision on Intel TSX - `mnist_deep` - size:784, batch:1

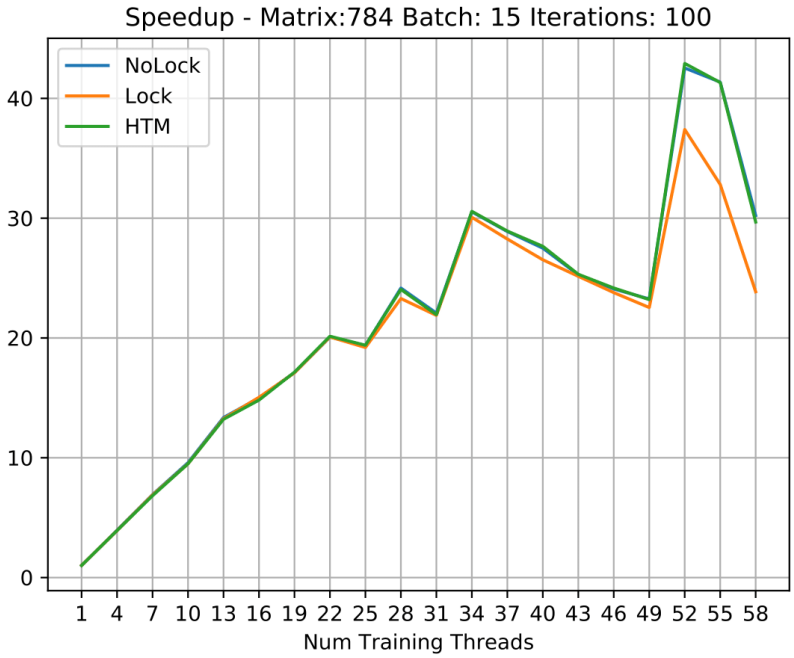

Figure 4.11. Speedup with HTM elision on Intel TSX - `mnist_deep` - size:784, batch:15

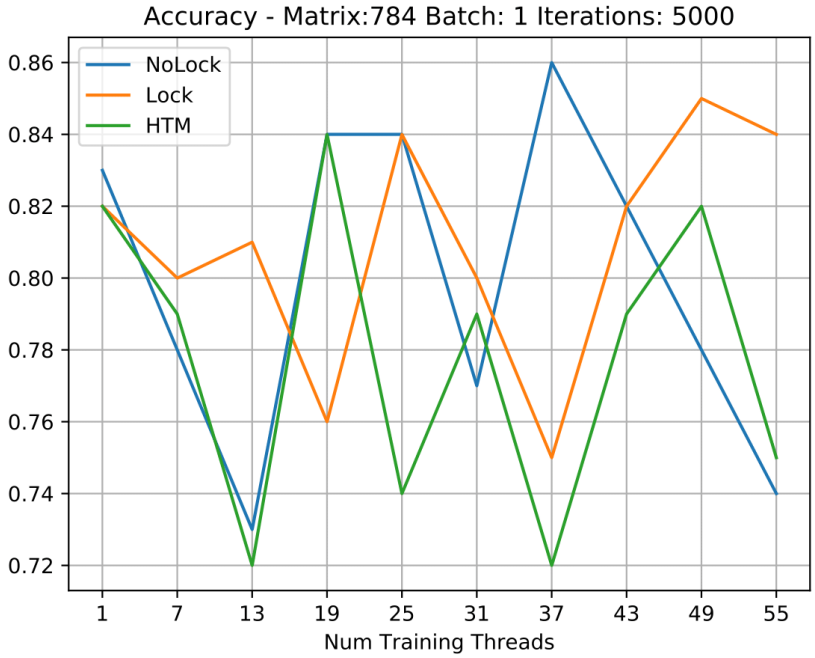

Figure 4.12. Accuracy with HTM elision on Intel TSX - `mnist_deep` - size:784, batch:1

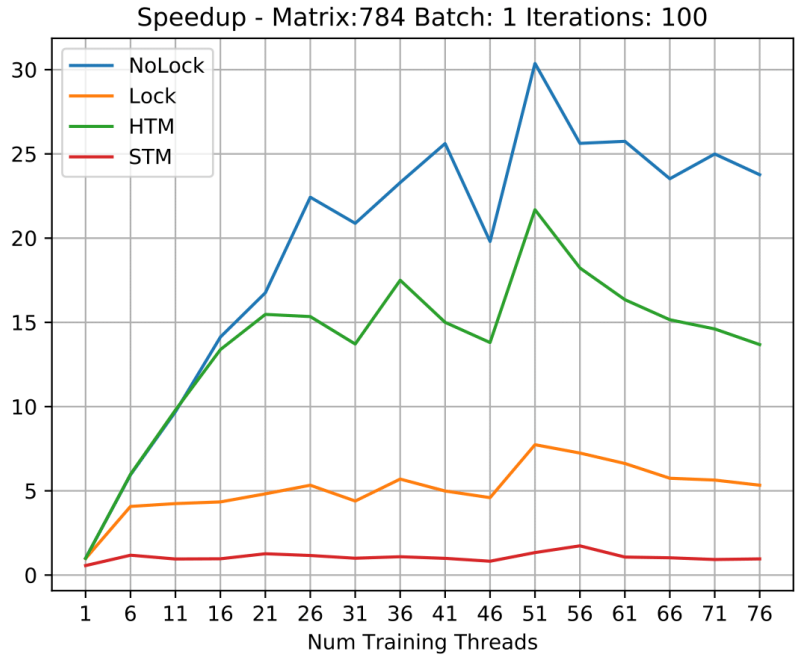

Figure 4.13. Speedup with STM - default c.m., size:784, batch:1

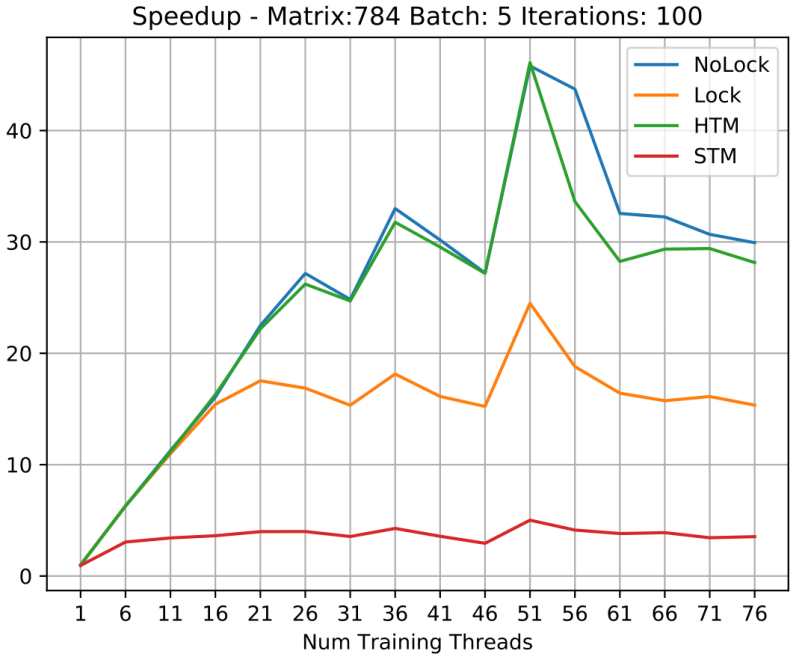

Figure 4.14. Speedup with STM - default c.m., size:784, batch:5

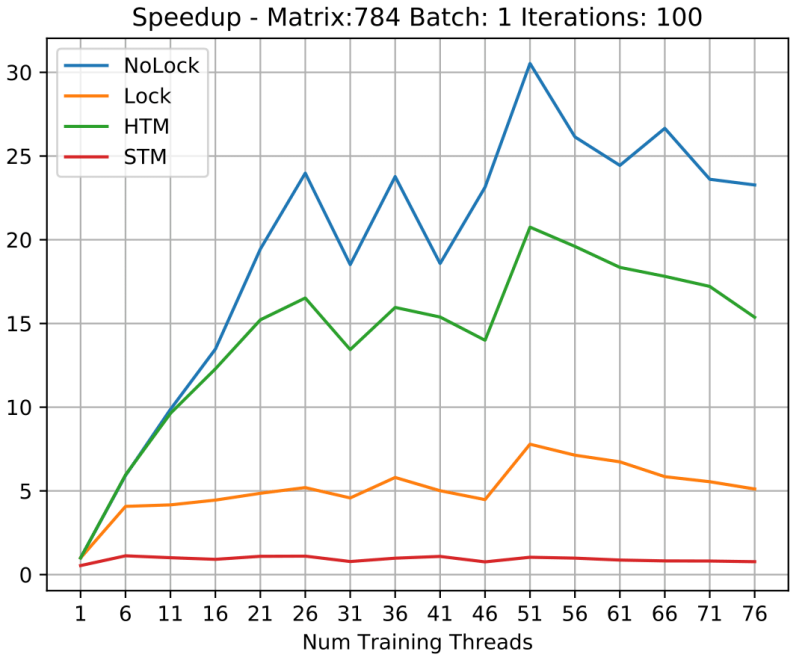

Figure 4.15. Speedup with STM - backoff c.m., size:784, batch:1

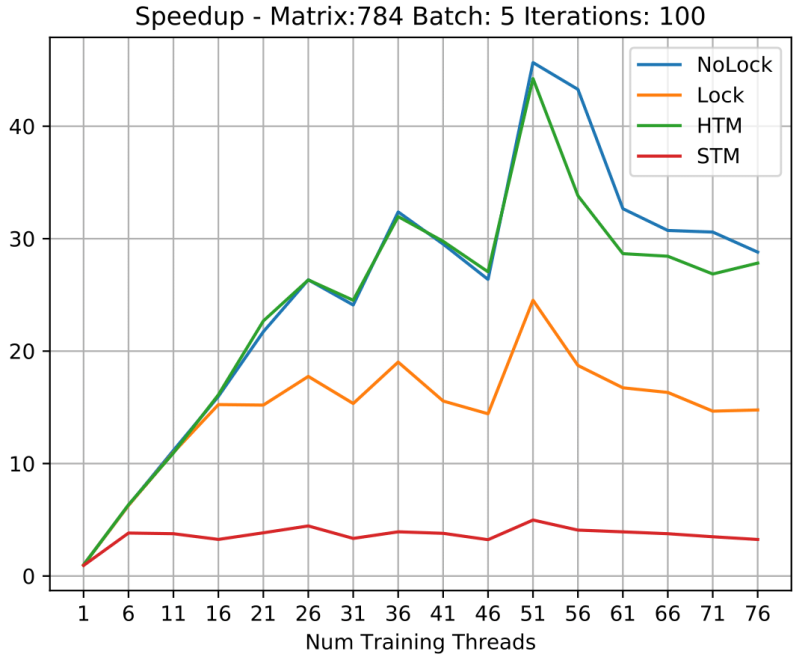

Figure 4.16. Speedup with STM - backoff c.m., size:784, batch:5

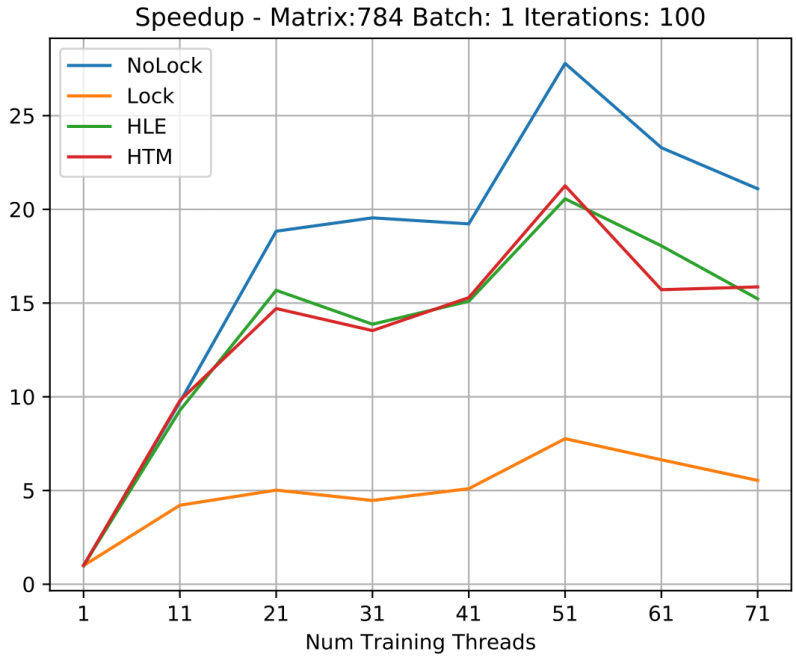

Figure 4.17. Speedup with Global TLE - default elision configuration

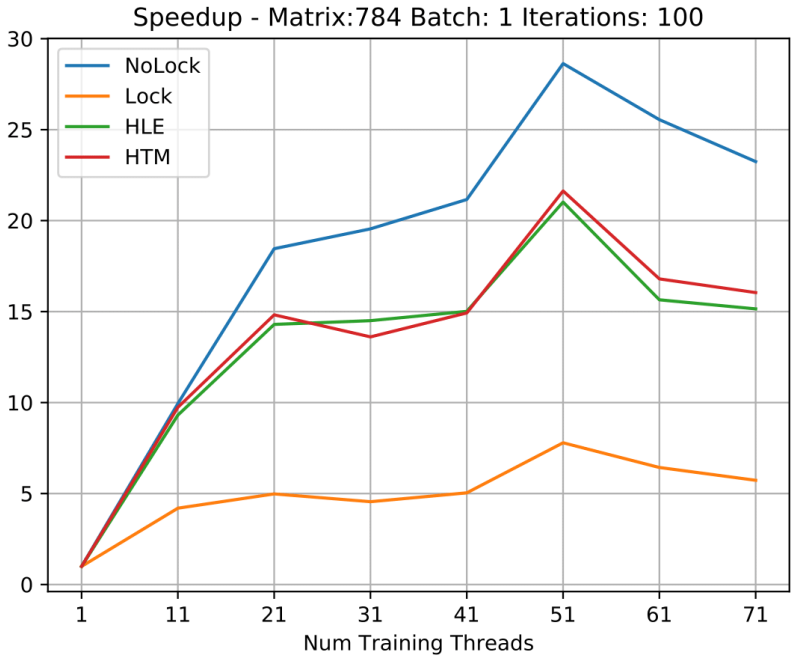

Figure 4.18. Speedup with Global TLE - custom elision configuration

Chapter 5

Conclusions

The objective of this thesis was providing an answer to the question whether the usage of the transactional memories, either in their software and hardware implementation, is convenient for training machine learning algorithms on TensorFlow.
Looking at the results of the tests executed to answer this question, it is difficult to provide a definitive answer. On one hand, there are clear signs indicating that the integration of hardware transactional memories in TensorFlow is convenient. In particular:

1. compared to the default lock-based implementation, the implementation of TensorFlow that features the hardware transactional memories requires a lower training time when the size of the batches read from the dataset is minimal (mini-batches)

2. on a POWER 8 machine, the usage of the hardware transactional support allows to achieve an average increment of the speedup around 25% in the training of the linear model, with a unitary batch size

3. on a machine featuring Intel Haswell, the usage of the hardware transactional support allows to increment the average speedup by two times in the training of the convolutional neural network, with a unitary batch size

On the other hand, it also seems pretty clear that:

1. it is not convenient to use software transactional memories for the update of the parameters of a model. The intrinsic overhead, due to the detection of conflicts and the maintenance of the read and write sets, represents a big limitation for their performances

2. there is no advantage in using the hardware transactional support to elide all the locks of TensorFlow with respect to focusing on the elision of the single lock that guards the application of the results of the gradient descent algorithm

Moreover, the results regarding the average accuracy of the two trained models don't show any clear difference between the lock-based and the lock-free scenario. A possible justification for this unexpected result, is that in none of the two models, although quite different, the synchronization of the application of the updates is not

necessary. This suggests that it is not possible yet to give a definitive response to the original question. Because of this, in conclusion, this work is to be regarded as a preliminary step that laid out the foundations for an innovative approach, based on transactional memories, to parallel training.

Bibliography

[1] Abadi, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, vol. 16, pp. 265–283 (2016).

[2] Abadi, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, (2016).

[3] Alpaydin, E. *Introduction to machine learning*. MIT press (2014).

[4] Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, (2015).

[5] Coates, A., Ng, A., and Lee, H. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 215–223 (2011).

[6] Dean, J., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231 (2012).

[7] Dice, D., Lev, Y., Moir, M., Nussbaum, D., and Olszewski, M. Early experience with a commercial hardware transactional memory implementation. (2009).

[8] Dice, D., Shalev, O., and Shavit, N. Transactional locking ii. In *DISC*, vol. 6, pp. 194–208. Springer (2006).

[9] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, **21** (2010), 1793.

[10] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 237–246. ACM (2008).

[11] Goldsborough, P. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, (2016).

[12] Hall, B., et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks (2017).

[13] Hammarlund, P., et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, **34** (2014), 6.

[14] Harris, T., Larus, J., and Rajwar, R. Transactional memory. *Synthesis Lectures on Computer Architecture*, **5** (2010), 1.

[15] Herlihy, M. and Moss, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM (1993).

[16] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An introduction to statistical learning*, vol. 112. Springer (2013).

[17] Kleen, A. Adding lock elision to linux. In *Linux Plumbers Conference, August* (2012).

[18] Langford, J., Smola, A. J., and Zinkevich, M. Slow learners are fast. *Advances in Neural Information Processing Systems*, **22** (2009), 2331.

[19] Le, H. Q., Guthrie, G., Williams, D., Michael, M. M., Frey, B., Starke, W. J., May, C., Odaira, R., and Nakaike, T. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, **59** (2015), 8.

[20] Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. Y. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pp. 265–272. Omnipress (2011).

[21] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86** (1998), 2278.

[22] Mitchell, T. M. *Machine learning.* McGraw Hill (1997).

[23] Nair, A., et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, (2015).

[24] Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M., and Tomari, H. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 144–157. ACM (2015).

[25] Rajwar, R. and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 294–305. IEEE Computer Society (2001).

[26] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701 (2011).

[27] Schrimpf, M. Should i use tensorflow. *arXiv preprint arXiv:1611.08903*, (2016).

[28] Shavit, N. and Touitou, D. Software transactional memory. *Distributed Computing*, **10** (1997), 99.

[29] Sousa, T. B. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, vol. 130 (2012).

[30] Stone, J. M., Stone, H. S., Heidelberger, P., and Turek, J. Multiple reservations and the oklahoma update. *IEEE Parallel & Distributed Technology: Systems & Applications*, **1** (1993), 58.