CHALMERS

*MASTER'S THESIS*

# *on* **Software Transactional Memory for Graphics Card**

*by* MUHAMMAD TAYYAB CHAUDHRY
*Department of Computing Science and Engineering*
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg Sweden 2009

THESIS FOR THE DEGREE OF MASTER OF SCIENCE

# Software Transactional Memory for Graphics Card

Muhammad Tayyab Chaudhry
mtayyabch@yahoo.com

**Supervisor:**
Philippas Tsigas
tsigas@chalmers.se

**CHALMERS**
*Department of Computing Science and Engineering*
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg Sweden
June 2009

## Abstract

The introduction of CUDA, NVIDIA's system for general purpose computing on their many-core graphics processor system, and the general shift in the industry towards parallelism, has created a demand for ease of parallelization. Software transactional memory (STM) simplifies development of concurrent code by allowing the programmer to mark sections of code to be executed atomically. The STM will then guarantee that other processes will see either none or all of the writes done in in that section. In contrast to using locks, STM:s are easy to compose and does not suffer from deadlocks. An STM can thus be seen as a concurrency control mechanism.

**Keywords:** Parallel processing, STM, Concurrent programming

# Preface

***The three great essentials to achieve anything worth while are, first, hard work; second, stick-to-itiveness; third, common sense.***
***-Thomas Edison***

I had an experience of a couple of years of web programming but latter turned to Computer Science teaching before I took admission in Masters program for Networks and Distributed Systems in Chalmers University of Technology. Here during studies I took two courses in Distributed Systems one of them was led by Philippas Tsigas. I made up my mind to do my thesis in Distributed Systems a year ago. Philippas welcomed me and gave me a brief introduction to Software Transactional Memory or STM and suggested to do some knowledge building by studying the related research papers. STM was a new field for me. Initial study led me believe that it will be a tough job but I was excited to take the challenge. As I kept on studying the recent research work, I realized that the future of Computer Architecture is Multi-core Systems which will definitely require a concurrency control mechanism for parallel programming. I finalized my thesis topic after consulting Philippas. But as a trial he supervised a 15 credit programming project on STM. After the research work studies, I was introduced with CUDA enabled NVIDIA's graphics card which contains hundreds of many-core processors. And then Philippas plainly told me to make an STM for NVIDIA's graphic card. It made me shiver for a moment because so far whatever I had studied was related to STMs for dual-core or quad-core processors. Whereas in addition to CPU, now I had to control more than a couple of tens of many-core processors on graphics card. It was a relief that CUDA is very similar to C language. I started working in device emulation mode, which at that time was sequentially executing. To my astonishment when I actually ran it in debug mode which runs concurrently, there was no error and no hang-up of system. It increased my confidence. At each and every step in this thesis, I climbed one step forward after each successful run. But this was not always the case. Whenever I was unsuccessful, I just tried to change the logic. I started with simple locking and with guidance of my supervisor, Philippas I could eventually run a test STM that proved to be working on shared memory. This prototype showed that it is possible to create synchronization among concurrent shared memory accesses. It had yet to be polished to add new features as I will explain latter and to provide ease of use to CUDA STM user. The excitement in working with CUDA is that its an unmanaged environment and I based the STM on C language pointer operations. Sometimes I really had to work hard just to find out that a pointer was going outside the desired range. But it did not lower my concentration ever. My project was approved by Philippas and he extended it further to mater's thesis making it over all 45 credit thesis. Much of the time was spent in searching, developing and testing different logics so that CUDA STM could become faster and generalized for maximum usage. I developed and programmed it alone under the supervision of Philippas. Where as other STMs are being developed by teams of professionals. This STM is the first one ever built in CUDA. I think this is one of the main reasons that kept me working hard with dedication throughout this thesis. I am glad that I could finally fulfill the requirements.

# Acknowledgements

I would like to pay my gratitude to a number of people who actively played their roles besides me, during this thesis. In particular, I would like to thank the following people for their support:

- My parents, **Mr. & Mrs. Rashad Tanveer Chaudhry**, for always encouraging me to acquire higher education and praying for me. Yes, they are proud of me.
- **Philippas Tsigas,** for having faith in me as my mentor for a whole year and agreeing to supervise my thesis. He encouraged me at every step to apply my own mental capabilities to develop logic and keeping me from going astray.
- **Daniel Cederman,** for his technical support that always led me to do more and more experiments polishing my work.

.

# Contents

# List of Figures and Tables

Chapter 1

# Introduction

Computer processor research have previously been focused on increasing the clock speed, but as of late the trend has shifted towards increasing the number of processors instead. This has lead to increased pressure for applications to become multi-threaded to take full advantage of the new computing power. But with increased parallelism comes the problem of efficient synchronization, since threads that concurrently access shared memory can easily corrupt data by accident.

The traditional way of synchronizing memory accesses have been to use mutual exclusion, using locks to only allow one process to access shared memory areas at any given time. However, this kind of lock based synchronization makes it hard to compose function calls and lead to problems such as deadlocks, where two processes are both waiting for the other one to give up a lock, and convoying, where a process that holds a lock gets swapped out causing other processes to wait unnecessarily long for that lock.

## 1.1 Transactional Memory

Transactional memory (TM) provides an alternative concurrency control that can eliminate these problems or at least minimize them. A TM allows the programmer to mark a section of the code that is to run atomically [1], i.e. it should appear to take place instantly to all other processes. The TM keeps track of all reads and writes in the code block and only stores the new data if no other process have interrupted it. If a transaction notices that another transaction has written to memory read in the transaction, the
transaction will be simply restarted.

### 1.1.1 Hardware Transactional Memory

Hardware transactional memory systems have modifications in processors, cache and bus protocol to support transactions [12].

### 1.1.2 Software Transactional Memory (STM)

Since there are no commonly available hardware transactional memory, most implementations are completely software based, so called Software Transactional Memory (STM). Purpose of a TM is to provide opportunity to programmers to achieve high degree of parallelism in applications without concentrating on the mechanism of synchronization.

## 1.2 Database Transactions

*Transaction* is in-fact a database concept. It consist of a single or a series of data read and data manipulation operations. At the end of transaction, if there are no database errors then commit the transaction other wise rollback all the steps and end it.

A transaction assures four basic properties for the correctness of operations and data itself. They are called ACID properties.

### 1.2.1 Atomicity

That a transaction is completed in full or not at all. If a transaction fails partially then it fails completely and should be restarted. It further refers to requirement of *recoverability* property which requires that no successful transaction should have read any data value updated by an unsuccessful transaction.

### 1.2.2 Consistency

It refers to the requirement that data is written back to database when it does not violate any of the consistency rules  and only valid data will be written back to database. The database is shifted from one correct state to other correct state after each transaction. If any of the consistency rules is violated, the transaction is rolled back and database is restored to a consistent state.

### 1.2.3 Isolation

It means that every transaction proceeds independent of any other transaction. It means that a successful transaction will not interfere with any concurrent transaction during its execution. Each transaction is executed as if it is the only active transaction. The highest level of isolation is the serial execution of all transactions. It is also known as *serializability.* Even if there are concurrent transactions, if the final outcome is equivalent to serial execution in any order, it is assumed that serializability is achieved.

### 1.2.4 Durability

The modifications performed by any successful transaction can survive system failures. It also refers that a committed transaction cannot be reversed. Even in case of system failure, transaction effects can be restored from database log. But in case of STM, this is not the case [2].

Transactions can be executed in a centralized database as well as a distributed database. In case of distributed data, a transaction may cover many hosts.

# Chapter 2

# Challenges for STM Transactions

A critical analysis by Felber et al. [2] indicates that it is non trivial to base STM model on database transactions.

## 2.1 STM Transactional Durability

STM transactions need not to be *durable* because their effects need not to survive the crash of the process hosting the transaction. This is a clear difference from database transaction whose effects must be preserved in case the active database user process crashes. Thus it is not the same durability as of the database transactions. It is sometimes argued that the memory transactions are only ACI.

## 2.2 Programming Languages

In terms of *programming languages*, unlike database transactions where each SQL statement is inherently executed as a single transaction and individual transactions can co-operate to perform a single transaction, in case of STM transactions the programmer has to carefully define atomically executing code block for transactional access. Any shortfall in concurrency analysis may lead to inconsistent results when a data item is concurrently accessed from a transactional and non transactional code. Database Management System can handle such conflicts. Without a properly designed STM runtime it is not easy for an application developer to check and develop an error free code whose execution is robust to such concurrent read / write accesses to a single data item.

## 2.3 Semantics

In terms of *semantics*, to prevent transactions from showing unexpected behavior and exceptions, a database requires its transactions to show the property of *serializability* means that each successful transaction can be independently marked in time as non overlapping with any other transaction which means that each transaction should produce the same result as if it was executed serially. Concurrent STM transactions may cause read-write conflicts which produce non-serialized results. STM runtime should be designed to implement the theory of *recoverability* to avoid reading inconsistent data which is being updated by concurrent transactions. There can still be a conflict if a transaction reads between two updates of successfully committed concurrent transactions that overwrite the results of the each other.

## 2.4 Transforming Transactional Code

Automatically transforming the non transactional concurrent code to transactions is a difficult task. It contradicts database traditions in which such a code inherently runs as a transaction. There are two alternatives to that, either separate the transactional code from non transactional one in which the latter code is not guaranteed for consistency or  dynamically categorizing  transactional and non transactional access to shared objects. This conflicts with traditional database concepts [2].

## 2.5 Implementation

Regarding the *implementation* of the STM transactions, the state management of a  transaction is difficult task in concurrent access. A challenging issue with STM  transactions is to differentiate between read access and write access to transactional data for data *contention* management. Even the use of *encapsulation* is not sufficient fro always separate read from write access. Read-only accesses create less conflicts than write-only accesses. Therefore early monitoring of such accesses is essential [2].

## 2.6 Serial vs. Concurrent Execution

Database transactions are mostly optimized for serial execution. But the STM transactions are actually meant to run on multi core systems to increase the concurrency and boost up the performance. Research on STM transactions optimization is still in progress [2].

# Chapter 3

# Concurrency Control Strategies

## 3.1 Simple Locking

Lock based concurrency control has been and is still used to synchronize concurrent accesses to shared memory by multiple threads. Each thread has to lock the shared memory before accessing it. When the memory operation/ s are successfully done, the lock is released. Till the lock is released, all the other processes trying to access the lock will have to wait. Such a mutual exclusion can be implemented by using *semaphores*. Lock is an abstract data type in programming languages like Java. A lock is either taken or is released. Locks can be shared but read-write collisions are most likely to occur with increase in number of processors. The simple solution is to use separate types of locks, one is for readers which can be shared and the other is for writers which cannot be shared. So there can be multiple readers at the same time with no writer. And only one writer in the shared memory with no readers.
There are various algorithms and various techniques to implement this. It is easy to code and easy to create new locks as and when required. But simple locking suffers performance loss due to *deadlock* when two threads wait for the availability of lock/s already held by each other. Simple locking is *non composable* as per James et. al [3] for example in a banking transaction to move amount from one account to other will require either locking of all the accounts or a single account. If a single account is locked then it may lead to deadlock between two transactions in which one transaction tries to move amount from first account to second and the other transaction tries to move amount from second account to first one. An alternative approach to acquire a special lock prior to access the two locations. As the number of objects increase, so does the number of locks and thus difficult to manage the process. There is a problem of *priority inversion* when a low priority thread acquires a lock and all high priority threads have to wait for that lock to be released. Thus a concurrency control run-time totally created on the base of locks will suffer from performance loss and difficult to generalize for all types of application.

## 3.2 Multi Version Concurrency Control (MVCC)

This is a non lock based concurrency control mechanism used in databases in which each transaction stores a snapshot of the database in the form of versions of objects that it reads. Transaction gets the exact copies of objects and updates them locally and at commit time, it validates that the versions of all objects read are still same, meaning that no other transaction has changed any object in the global memory. Only after the full read set validation, the updated local objects are copied back to global objects and their version numbers are increased serially. This ensures the important property of *serializebility.*
It is however notable that no transaction should read a version number that has not yet been produced; meaning that a transaction should only read version numbers produced by successful transactions to

ensure *recoverability.*
Also when a transaction has to write some object, then it must have read its version number before or read it prior writing. A transaction in MVCC is *recoverable* if all the transactions which produced version numbers that it has read have already committed; otherwise the transaction commit is delayed.
Object versions can be marked with time stamps. The basic idea is to make every transaction commit with unchanged versions of its read set objects so that it can be marked distinctly in history [4, 5]. MVCC tries to implement the distributed system total ordering for centralized database and partial ordering for distributed database [4]. In a database management system(DBMS), many versions may be stored for a single object depending upon how many concurrent transactions have read it and how many commit, but each transaction runs as if there is only one version of that object is kept by DBMS.
MVCC can be used with *Two Phase Locking (2PL)* [5]. In this case there are two versions of each object one is known and readable to all concurrent transactions while the other one is the updated version which is published as soon as the writer transaction commits successfully. During the execution, a transaction acquires three types of locks namely Read lock, Write lock and Certify lock. Function of Read lock and Write lock are obvious while Certify lock is used to delay a transaction from writing over an object while there is at least one read lock over it.. These locks are convertible from read to write to certify. There is a chance of deadlock to happen which can be avoided by using traditional methods like cycle detection. A transaction must obtain all certify locks on all objects that it tends to write over before committing. This is called two version two phase locking(2V2PL). Benefit of this locking technique is it provides less delay in Read than simple two phase locking(2PL) as certification is delayed till atleast one read lock is present on the same object while in 2PL a lock is only free when a transaction is terminated and other readers have to wait.. Notable thing here is that all version control and locking is handled by DBMS and transaction just proceeds sequentially.

## 3.3 Optimistic Concurrency Control (OCC)

Simple locking concurrency control approach does not give a reasonable gain in performance for example when there are less chances of conflicts in concurrent transactions. Simple locking can be called a *Pessimistic Approach* [15] when locks are acquired in the early stage of transaction and are not released till the end. This shortens the resource availability [15].
OCC is an alternative to pessimistic approach [6, 15]. This approach believes that not all of the transactions are conflicting therefore the locks should be acquired just before the memory access and should be released as soon as possible. The memory operations are performed in three steps:

1. **Read**: Read the data from the memory location and store it as a local copy. Perform the operations on it. And maintain a log of all changes.
2. **Validate**: When the process has completed editing of the data, it checks if in the meanwhile no other transaction has changed the value in the original memory location. This can be done by checking the logs of either previous transactions *(Backward validation)* or by reading the logs of the currently executing transactions *(Forward Validation)*. If there is a conflict in values read in the Read phase and the Validate phase, the process is restarted and we say that the transaction has aborted and will be restarted.
3. **Write**: If there is no conflict what so ever, the updated data is written in the original memory location. This is also called that the transaction has committed.

OCC has taken the newer approach that it is up to the reader to check at the end of the transaction that other threads have not changed the values that it has read in the past. For this, a transaction log is maintained and every read and write operation is recorded in it for future validation. A transaction effects can thus be rolled back using the log. The optimistic concurrency control gives independence to each thread and allows it to do its tasks regardless of what other threads are working upon. The limitation of this approach is that it is applied when there is a low contention, meaning there are less chances of conflicts because although the validation phase takes less time but locks are obtained for a very short time just before read and then released but write locks on the other hand are obtained just before write but are retained till the transaction commits. This short duration lock acquiring lowers isolation level and the transaction is rolled back if a to-be acquired lock is not available [15]. But OCC provides better resource availability.

Chapter 4

# Types of STM

STM can be classified based on different parameters. Transaction granularity refers to the level of concurrency. In a *Word based* STM, the memory is divided into some predefined blocks called "words". Memory access in this STM requires word address in the memory [2]. This STM provides concurrency at the level of data members of object. There are lesser conflicts to occur if two transactions access different elements of same object. An *Object based* STM assumes that the memory is divided into objects of various sizes. The objects also requires meta data [2] for example the properties and location. The meta data can be kept external or within the object. The memory access to the objects requires the base address of the object as an additional parameter [2]. A STM will be classified as *Time based* if it uses a global clock to ensure the consistency of data. They employ optimistic read operations (i.e., read operations are not visible to other transactions) because invisible reads are less expensive than visible reads. Time based transactional memories then guarantee on the basis of their time base that the snapshot that a transaction takes of the transactional memory at runtime is always consistent [5]. On the basis of updation policy [3, 7] a STM will be performing *Deferred updates* if it clones data objects to local copies then locally performs computation and then detects any conflicts and if no conflict is detected then writes back to shared memory. This type of STM uses read versioning for concurrency control and is lock free. Opposite to this is *Direct updates* in which a transaction directly updates shared data instead of local copies of objects. This STM has to maintain an undo log to restore shared objects in case a transaction aborts. In this STM, locks are to be used to prevent readers and writers to access an object which is being updated by a particular writer. Performance results show that a lock based direct updates STM performs better than deferred updates STM [3, 7]. Further, the STM can be divided into two types, based on the environment as *Managed Environment* STM in which applications access resources through an intermediate layer such as a Java Virtual Machine [8]. In *Unmanaged Environment* STM application code directly accesses the resources provided to its process [8] such as C/C++.

Traditionally the compilers assume object based STM for managed environment and word based STM for unmanaged environments. The only obvious reason for that is an object is a continuous series of memory chunks and the size of every user defined data type object varies and in an unmanaged environment, it is not possible to judge which memory location belongs to which object and since any memory location can be accessed by the code for example in C/C++, there is always a chance of data corruption. Although it cannot be proven that either a word based STM is better or an object based STM but the unmanaged environment can also accommodate the object based accesses with the help of compiler support to gain the performance benefit [8].

STM maintains the information for every object for example the locks in order to control the concurrency. This can also be termed as meta data or transaction log. Depending upon the design of STM, if it is a word based then the memory locations or words belonging to same objects are mapped using a hash function; or if it is an object based STM then the meta data is either external (linked to the base address ) or in-place (embedded into the object memory space if the object is not too large). Each scheme has its own advantages and disadvantages. For example in the object based STM, the in-place approach is beneficial as it reduces the cache footprint and there is only one cache miss for both meta data and the object when the read data is modified by another transaction. In case of external meta data, the object can be partitioned implicitly to avoid acquiring several locks in order to write to a single object. This partitioning is not possible in case of fixed object to meta data mapping in case of in-place meta data and this is a disadvantage when there is a high contention. Further it increases the cache footprint by increasing the size of already large objects. The in-place design also suffers the overhead of maintaining the object in the memory until there is no other transaction accessing it. This is due to the fact that meta data is to be remained type safe and since meta data is embedded in the object, so whole of the object is to be kept alive [8].

## 4.1 Object based STM in Unmanaged Environments

Using an object based STM in unmanaged environment requires that a programmer must use explicit calls to STM unless there is a proper support by the compiler to decide whether an access to a certain memory location as an object is safe or not. In other words, to automatically transform the memory accesses to STM runtime calls, we must make sure that the call is for an object in the memory identified by the base address of the object in memory. The compiler must identify that a certain memory location belongs to an object no matter if it is dynamically allocated or is on a stack or global and if so, then which object. This is however not an easy task since in unmanaged environment like C/C++ the programmer can access any memory location, casts between the types and pass pointers to the fields of an object as function parameters. Therefore an inter procedural analysis [8] is required that computes a points-to graph for the complete program. The *Data Structures Analysis* (DSA) can be used for this purpose to ensure context sensitivity (the data structure is uniquely identified based on call graphs ), unification (each pointer targets at most one location in the points-to graph ) and field sensitivity (distinguishing between different fields in one data structure). The DSA is embedded in the LLVM compiler framework.
The whole procedure is done in four steps:

1. Compiling and linking of the whole program using LLVM into a single module.
2. Use DSA analysis of this module to distinguish the object-based accesses from word-based accesses.
3. Identify the data structures involved in object-based access to the memory.
4. Transform the transactional parts of the application by choosing between the object based and word based accesses according to the DSA results.

After the DSA, the data structures are represented as nodes in the points-to graph. This graph is built incrementally by first analyzing each function and determining properties of nodes based on how pointers and data structures instances are used. For example if a pointer points inside somewhere to the memory region occupied by a node, then that pointer is assumed to be pointing towards the node. There are certain flags of information stating if the data structure is located on stack or heap. An additional information is provided telling if all the calls to that data structure have been identified or not. After that the caller and callee graphs of the program are merged to identify the nodes which have escaped. Information about those nodes is marked as incomplete. Further, if there is an uncertainty about two different type of pointers pointing towards same node, then that node is said to be collapsed. Some nodes are identified as external if they call some external function. A node is said to be accessed in a type safe manner if it has been completely analyzed, is not external and is not in an array. The compiler then transforms the transactional memory accesses to STM runtime calls. The location of the object is

determined from the base address and the size is determined by the type of the user defined data structure. For the objects with in-place meta data, sufficient space is allocated and the STM will fill the meta data.
At this stage, the word based and object based accesses are identified keeping the point that only the type safe accesses identified during the DSA are considered. When the external meta data is used, the lock word is updated before releasing the object. There is no delay in it. But when an in-place meta data is used, the lock is not released until the time stamps of all the transactions become greater than the transaction who last committed the particular object. This is due to the use of time based STM in this compiler optimization.

## 4.2 McRT STM

McRT [7, 9] is an experimental multi-core runtime and McRT-STM is an experimental STM  built on that runtime. It implements transactions using strict two-phase locking and contains commit and abort sequences that are blocking. It uses a strict two phase locking protocol to implement transaction manager. Each memory location is mapped to a lock and all locks are to be acquired before committing a transaction. Any transaction can abort other transaction if former is waiting for a lock acquired by the latter and the latter is not active. Blocking implementation reduces the number of aborts and memory management is simplified. Deadlocks are detected by creating a graph of waiting transactions and aborting one of the transactions if there is a cycle detected. Deadlocks can also be avoided by waiting for a finite amount of time for a lock to be released and then aborting. Data contention is avoided by imposing read / write  locks and a transactions must wait for the lock to be released and then acquire the respective lock and completes its operations. Only the transaction executed by a thread which is still holding a lock but has yielded the processor can be aborted by the current transaction which is waiting for that lock to be released. Locking mechanism is implemented by using either Reader-writer locking or by using a combination of read versioning and write locking. When using the Reader-writer locking a transaction has to read first before writing to that memory location. This means that it should first acquire read-lock and then update it to write-lock. A single 32 bit integer is used as the read-write lock. The last three bits are used for the locking mechanism. A Reader 'R' bit is to be set prior to reading so that a writer cannot update the memory location. A Notify 'N' bit is set when a reader intends to read the memory location already locked. An Upgrade 'U' bit is set if the reader intends to upgrade the lock to write. A writer can only acquire the lock if all the three bits are zero. When a writer acquires a lock, it stores a transaction pointer in the lock word with last three bits set to zero. When the reader acquires the lock, the top 29 bits store the number of concurrent readers. In read versioning, the lock word contains the version number of the memory location or the transaction descriptor pointer of the writer. A reader reads when the R bit is one, meaning that the version number is stored in the lock word. The reader stores the respective version number and upon commit checks the version number again and commits if the number is unchanged. The writer stores the version number before it updates the memory and increments it after writing. The readers use the n bit for the signal. The difference in this scheme is that the wait locations are guarded by a mutex to prevent race condition between readers and writers. The U bit is not used. By the experimental results, it is observed that the read versioning performs better when the number of processors are not high. By the experimental results, undo logging performs better than write buffering because write buffering has the overhead of searching for the most recent value from the buffer every time. For the managed environments like Java it is easy to implement the object based conflict detection but for unmanaged environments like C/C++, McRT puts the objects in predefined object segregation memory chunks according to the size of objects and uses two approaches. First one is to place the locks in-line with the object. This will improve the locality on the cost of cache space if no transaction is using the lock. The second scheme is to separate the locks

and place them externally. This benefits in reducing the memory wastage at the cost of worse cache locality.
McRT-STM API is created based on the above analysis. It contains functions to perform tasks like initiating a transaction, maintaining the dynamic nesting length, mapping an address to a lock, reading and storing the version number, acquiring the read lock, acquiring the write lock, storing the old value of a memory location before writing the new value to maintain the undo log, validation of version number, committing a transaction with validation and releasing of locks, aborting the transaction with roll back from undo log and some other related functions.
Some useful data structures are provided in the STM. For example the transaction descriptor for storing the meta data about the particular transaction, the read log and write log.

### 4.2.1 New Language Constructs Introduced in Java for McRT STM

New language constructs are introduced to implement transactions in Java[9]. They are as followed:

1. *atomic{S}*: Executes the statement S as a transaction. The memory effects in S are serializable with respect to the memory effects in other transactions. The effects of S become globally visible when it completes (either normally or exceptionally).
2. *retry*: Can only be executed inside a transaction block. It will block a thread and restarts its own transaction when an alternative path is available. This can also be used for condition synchronization since it can make the current transaction to wait because it cannot proceed due to the current contents of memory till another transaction changes them. It is similar to Java's *wait* statement except that there is no construct that could be equivalent to *notify* statement.
3. *or else*: It gives two alternative transactions that can be set in the manner ***atomic*{S1}** ***orelse*** **{S2}** executes S1 as nested transaction and if it is successful the operation is complete otherwise if S1 needs to block, it must call *retry* to start the S2 as alternative transaction. The memory effects of S1 are discarded. The *retry* blocks only the outer level transaction. This also requires the capability of partial undo which is provided.

   The second and third statements are used to compose the transactions and allow nesting of transactions.

   1) *Tryatomic*: Tries to execute a block atomically but fails if it tries to block.
   2) *When*: It blocks until a condition is not true, after that it executes a corresponding statement atomically.

Only the Java library functions can be called from inside the transactions and there is no direct interaction between the transactions. Effects of all atomic blocks are serializable.

## 4.3 Herlihy et al.'s Dynamic STM (DSTM)[3, 10]

DSTM is a deferred-update STM which detects conflicts at object level. Due to deferred updates, a transaction can access the object whose clone is being used by another concurrent transaction. This will result in a conflict which is resolved by aborting one of the transactions. DSTM system atomically replaces the old object in a with its modified version on successful commit. Before commit two conditions are verified. The first is read set validation, meaning that the versions of the read objects are still same. The second condition is that the current transaction is not trying to modify an objects which is being modified by another transaction. DSTM allows only one transaction to modify an object at a time. If there is a write-write conflict between two transactions, then one of them is aborted and restarted.

Chapter 5

# CUDA

CUDA™ (Computer Unified Device Architecture) was introduced by NVIDIA in November 2006 as a general purpose parallel computing architecture that implements parallel computing engine on NVIDIA GPU making it more efficient to execute computationally complex problems than on CPU. CUDA enabled GPUs have hundreds of cores which can run thousands of threads. CUDA applications include medical imaging, natural resource exploration, image recognition and in this case, CUDA STM.

## 5.1 CUDA Architecture

The CUDA Architecture [11] consists of several components:

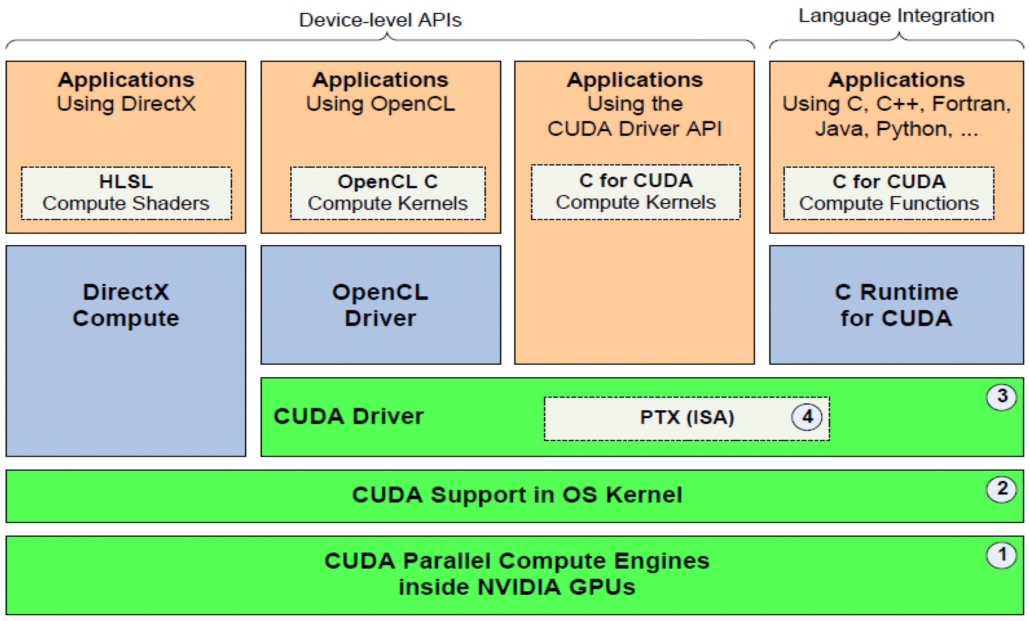
Device-level APIs
Language Integration
Applications Using DirectX
HLSL Compute Shaders
Applications Using OpenCL
OpenCL C Compute Kernels
Applications Using the CUDA Driver API
C for CUDA Compute Kernels
Applications Using C, C++, Fortran, Java, Python, ...
C for CUDA Compute Functions
DirectX Compute
OpenCL Driver
C Runtime for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines inside NVIDIA GPUs
1

Fig.1 CUDA architecture (courtesy of NVIDIA Corporation) [11]

1. Parallel compute engines inside NVIDIA GPUs
2. OS kernel-level support for hardware initialization, configuration, etc.
3. User-mode driver, which provides a device-level API for developers.
4. PTX instruction set architecture (ISA) for parallel computing kernels and functions.

## 5.2 CUDA Programming Interfaces

Two programming interfaces [11] are supported by CUDA Software Development Environment:

1. A device-level programming interface, in which the application uses DirectX Compute, OpenCL or the CUDA Driver API directly to configure the GPU, launch compute kernels, and read back results [11].
2. A language integration programming interface, in which an application uses the C Runtime for CUDA and developers use a small set of extensions to indicate which compute functions should be performed on the GPU instead of the CPU. This programming interface enables developers to take advantage of native support for high-level languages such as C, C++, Fortran, Java, Python etc [11].

To develop CUDA STM, the device level programming interface was used. It is provided toolkit containing the nvcc C compiler, CUDA runtime driver and other components.

Software environment of CUDA allows an enhanced version of C language to be used as high level programming language.

## 5.3 CUDA Language Abstractions

There are three key language abstractions of CUDA.

1. Thread groups hierarchy
2. Shared memories
3. Barrier synchronization

These abstractions allow the programmer to divide a problem into sub-problems and solve them in parallel on any number of processor cores. A C function that is to be called from CPU to begin parallel execution on GPU is called *kernel*.

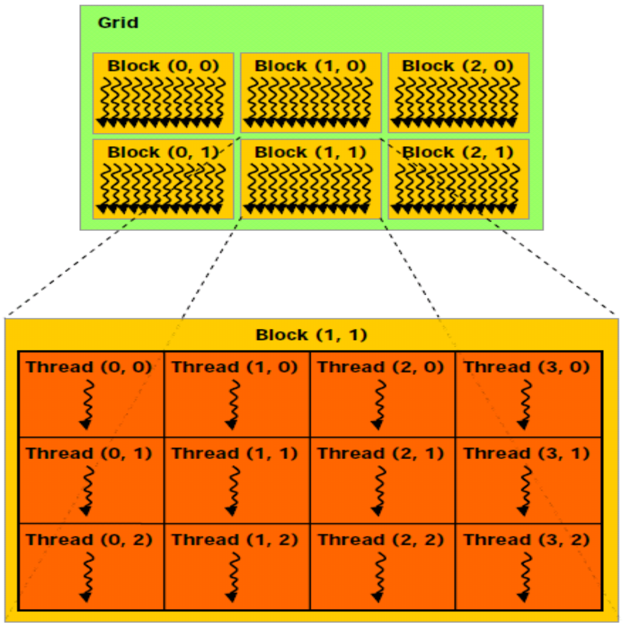

Fig.2 Grid of Thread Blocks (courtesy of NVIDIA Corporation) [13]

*Thread group* hierarchy consists of thread blocks and grid. The term thread block refers to a set of threads having its own unique id and running on the same multiprocessor. Multiple thread blocks can run on same multiprocessor. Each multiprocessor has 8 scalar processors each running one thread on it. The instruction set is SIMT (Single Instruction Multiple Threads). Each thread executes independently. The SIMT unit creates and executes a group of 32 parallel threads called *warp* at a time. There is however a limitation on the maximum number of threads in a block of up-to 512 threads. Individual thread blocks can be arranged to form a grid on the cost of reduced communication between threads due to separate thread blocks. Here an important distinction is to be made which is, a thread block can contain a single or hundreds of threads. When thread blocks are joined to form a grid, depending upon the dimensions of each block, the total threads can be equivalent to number of blocks to thousands. A single kernel is invoked for whole grid at a time.

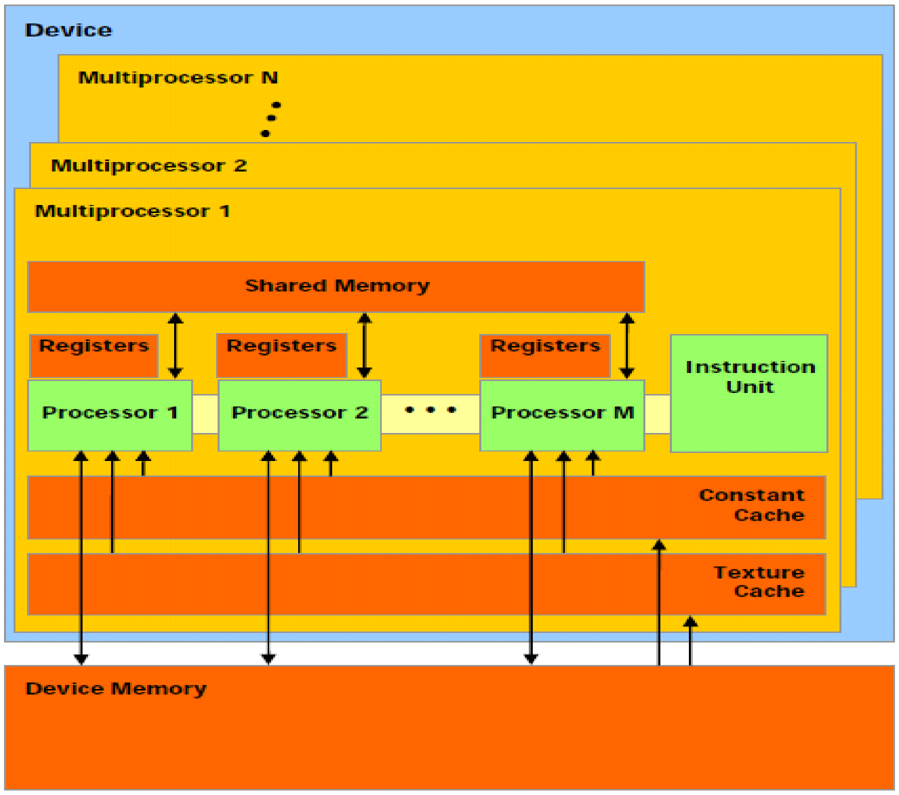

Fig.3 A set of SIMT multiprocessors with on-chip shared memory (courtesy of NVIDIA Corporation) [13]

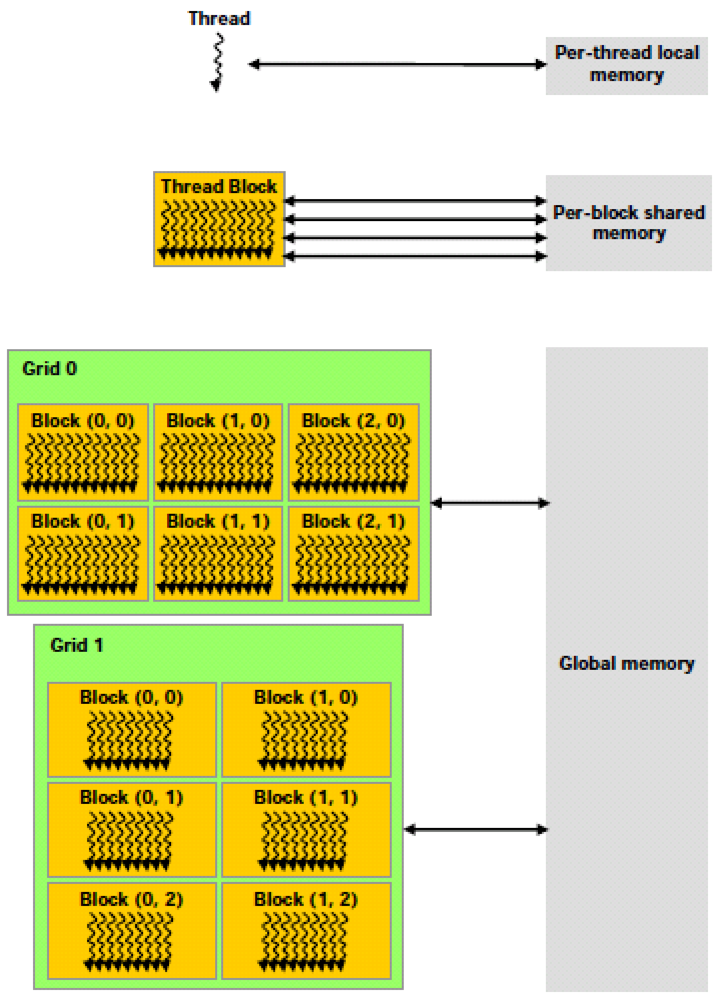

Fig.4 Memory Hierarchy (courtesy of NVIDIA Corporation)[13]

Threads of same as well as different blocks can share memory and can perform read and write operations on memory hierarchy area called global memory. There are two other memory hierarchies namely texture memory and constant memory but they are read only. On the top of the hierarchies is the block level read and write shared memory. Barrier synchronization holds the threads till they are allowed to proceed.
Programming model of CUDA allows a coarse-grained task division among many-core GPU processors with a further possibility of fine-grained task division among threads of each block.

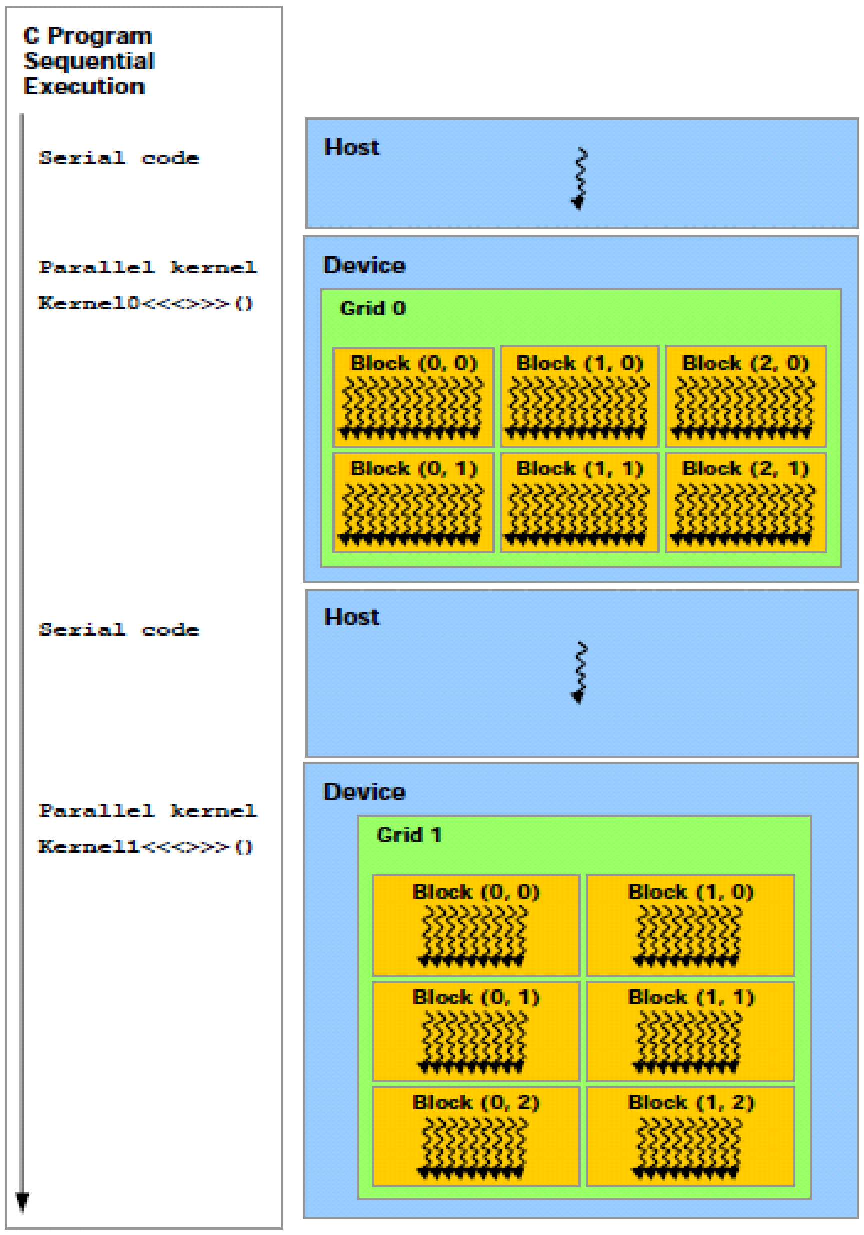

Fig.5 CUDA Heterogeneous Programming (courtesy of NVIDIA Corporation)[13]

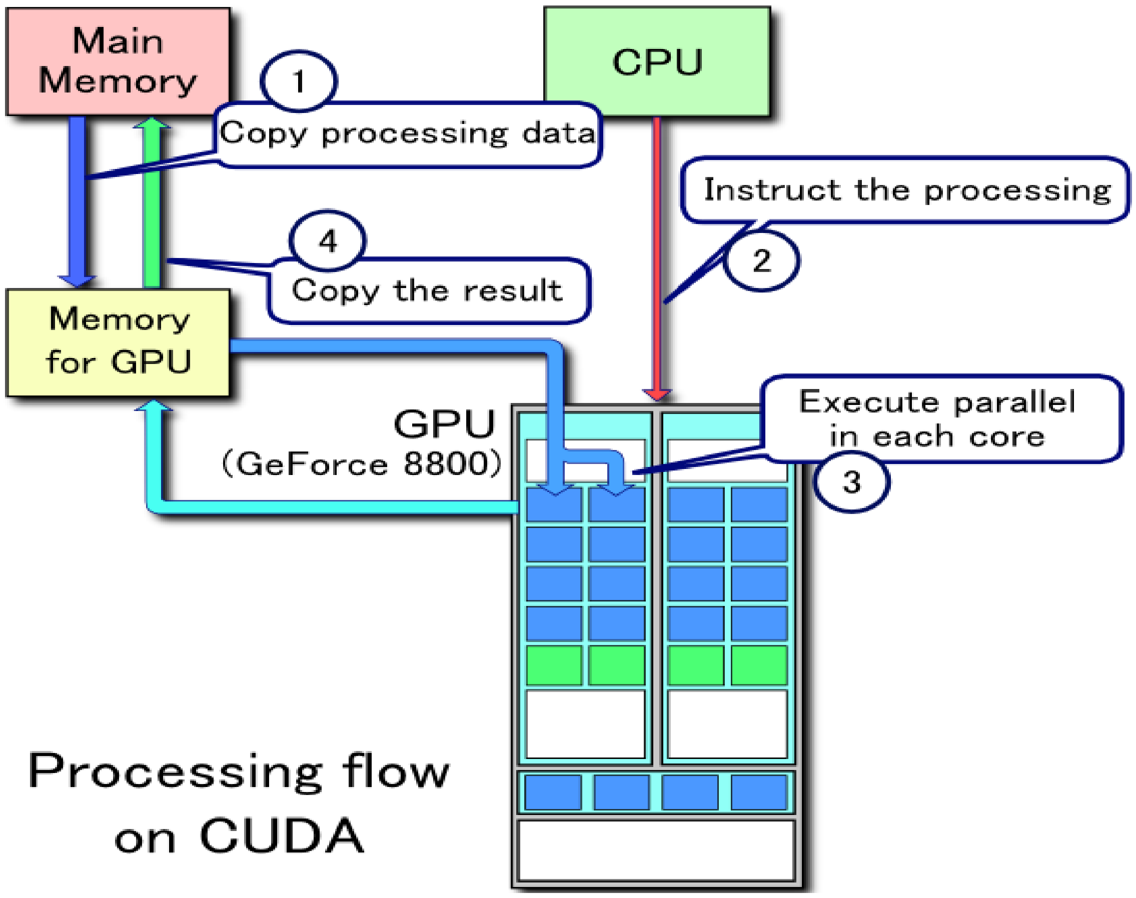

Fig.6  Processing flow on CUDA [14]

In order to process data concurrently on GPU by using CUDA, following four steps are to be performed:

1. Copy the data from main memory to GPU memory by using CUDA commands.
2. Execute kernel from CPU on GPU and pass necessary parameters to kernel along with grid configuration.
3. GPU executes kernel concurrently on all thread blocks.
4. After the execution, the processed data has to be copied back to main memory using CUDA commands.

## 5.4 Limitations of CUDA for a STM

There are however certain limitations in the development of STM in CUDA. These are listed below:

1. The language is not an object oriented language and there is no generic data type available to generalize an application for any type of data structure.
2. The environment does not allow to debug the code with concurrent access except in device emulation mode which is a sequential access mode and not compatible with actual run.

3. Busy loop or wait or indefinite loop cannot be executed for more than a fixed interval of time because it will hang the whole system which has to be rebooted then. This is the most important factor in case of STM.
4. The powerful GPU environment gives quite different results than device emulation mode.
5. There is no support from the CUDA compiler to define atomic code blocks.
6. For memory manipulation through pointers, only the global memory can be used across various internal functions defined for STM.
7. CUDA does not support recursive functions. Therefore binary-tree and skip-list are difficult to create.

Chapter 6

# CUDA STM Design Motivations

CUDA provides a realistic and practical environment for parallel processing and provide excellent motivation to develop a concurrency control mechanism. No other platform provides hundreds of cores at a time for parallel processing than CUDA. We used it to implement STM for concurrency control to shared memory. This is the first STM build ever in CUDA and it can be used as word-based as well as object-based or a mixture of both.

Following are the steps and decisions taken while designing CUDA STM:

Initially I chose the STM to work as object-based STM. Since objects are not possible in CUDA, I based the concurrency at data structures level. Latter on I extended it to word-based STM. Still next step was allowing the mixture of both approaches. This is further explained latter.
In order to decide update method I chose deferred update because in highly parallel CUDA environment where hundreds of concurrent transactions may be executing, a transaction can perform relatively more isolated if provided a local buffer to perform its operations on object clones and latter on perform the conflict detection.
In such a highly parallel execution environment, locking a memory areas by following the pessimistic approach reduces the availability of memory resources. Therefore by following optimistic concurrency control (OCC) [15] to lock memory for a very short time increases the availability.
Here an important thing to note is that all the blocks execute the same kernel at a time and operate on the shared memory. In other words there is a high contention of data. OCC is useful in low contention and less conflicting environment and in case of CUDA, it will simply increase conflicts.
This also leads to a thought that if pessimistic locking is used, it will make other blocks to suffer while a memory area is locked and loss of performance because the GPU can be used for a limited amount of time. Any how I decided to use OCC because it uses fine grained locking; for example in creating a binary tree, a transaction starting from root node will only need read access to child nodes while parsing and a write access to the leaf where the new node will be inserted.
OCC requires a transaction log and a transaction buffer because a transaction must keep the record of every memory access. Latter on in the end of transaction, some locks are to be acquired based on these records. Buffer is required to keep local clones of the objects. Transaction log is explained latter along with buffer.
To copy a data structures, I tried to impose a limitation that each data structure must contain a method which copied every element to another memory location pointed by a simple pointer of same data structure type. I found it restrictive in use since each data structure had to be designed like that to be usable with in STM.
While searching for an approach to make STM generalized for any type of data structures, I required a generic data type for type casting. Since there is none in CUDA so far, I decided to copy the data structures byte by byte using simple char pointers. This made STM generalized as well as I got rid of previous problem of restricted design of data structures. Copying data in this manner can lead to fall in

performance but this is the best choice available.

Any concurrency control mechanism requires validation of data reads before computed results can be written back to shared memory. The read data must be most recent on which computed results should be based and then written back. If any other transaction has updated at least one of the data items from read set of another transaction, then the computed results of current transaction are no more based on fresh data at commit time. So far I had no method to verify any change in data in global shared memory.

Multi-Version Concurrency Control (MVCC) [4, 5] provides a simple technique to check any change in data. It is to associate a version number with every data item. Now this approach also uses locking in the background to refrain a writer from overwriting the data while it has at-least one reader.

OCC uses two types of locks and MVCC uses three types of locks as explained in the previous sections. I had total five locks. I decided to use a shareable read lock and a non shareable write lock and a version number with every data item and use transaction log to keep the record per access to memory. Regarding the buffer, same problem of no generic data type present in CUDA led to use a long array of character data type and copy clones in it byte by byte as explained before and type cast it to appropriate data type at runtime.

Till this step, locking was done by using a single integer as lock. Inspired by McRT [7, 9] a reader checks the availability of lock if lock value is zero or greater than zero and increasing it by 1 indicating the number of concurrent readers and on leaving the reader decreases the lock value. Thus read lock is shareable. A writer can only access the lock if it is equal to zero and changes it to -1 indicating the non availability of lock to any reader or any other writer.

Now for the validation phase, in addition to verifying the version number of each data item in read set, I had to lock them as well. A high isolation is provided if all the data items go into write lock to prevent any other transaction to access them and it will very well ensure serializability on the cost of reduced availability of memory resources. I created a binary tree which could have 300 nodes in 4927 milli seconds.

At this stage, a fine grained locking was being used. It is fine grained in the sence that locks were being acquired for a very short period of time by following OCC. Further analysis to increase the performance led to experiment with MVCC also. Here a notable thing is, by acquiring write locks over data items that were accessed read only will make them under go an *illusive write-lock* which is not useful. If those data items are locked using a read-lock, still it is of no use because they will not be read and any writer transaction will suffer non availability of lock.

MVCC on the other hand uses three types of locks as explained before in section 3.2. An experiment was performed to make CUDA STM use minimum locks because locks are the major source of conflicts. Locks could not be avoided on write access because after all if a transaction is reading a data item and there is no locking for another writer transaction then reader will get a corrupt data value, partially overwritten by the writer or there may be a confusion in version number being not updated by writer and reader meanwhile completes reading. Therefore write lock is unavoidable. While the read lock can be avoided if a proper algorithm was there.

It was performed with simple algorithm of storing version number of lock and then checking the lock vale not to be in write mode before reading. If it is not in write mode then start reading and after reading, compare the version number before and after read. If the version number is same then its a valid read. Nothing is read if version number does not match or the lock was in write mode. When this approach was tested, there was a remarkable increase in performance in terms of total time. It is shown in following tables and figures. The last two columns will be explained latter when multi-threading will be explained. Following are the test results performed for comaprison of different concurrency control mechanisms.

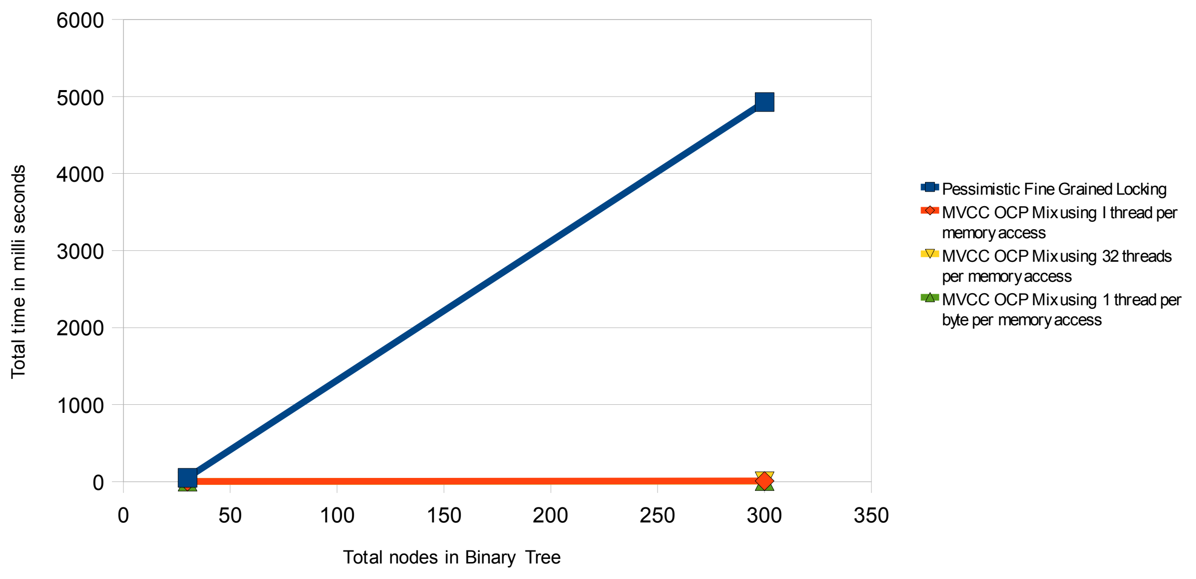

Fig.7 Performance graph for different concurrency controls for Binary-tree

| **Total Nodes in Binary Tree** | **Total time in milli seconds using Pessimistic Fine-grained Locking** | **Total time in milli seconds using MVCC & OCP Mix using 1 thread per memory access** | **Total time in milli seconds using MVCC & OCP Mix using 32 threads per memory access** | **Total time in milli seconds using MVCC & OCP Mix using 1 thread per byte per memory access** |
|---:|---:|---:|---:|---:|
| 30 | 50 | 1.2 | 1.35 | 1.39 |
| 300 | 4927 | 7.9 | 4.75 | 5.6 |

Table 1 Performance in milli seconds for different concurrency controls for Binary-tree

The above table and figure shows that with the growth of binary-tree, the Pessimistic approach shows worst performance while the combination of MVCC and OCC shows a healthy performance gain in time. This performance is improved with the used of multi-threading in CUDA STM as explained in section 6.6.

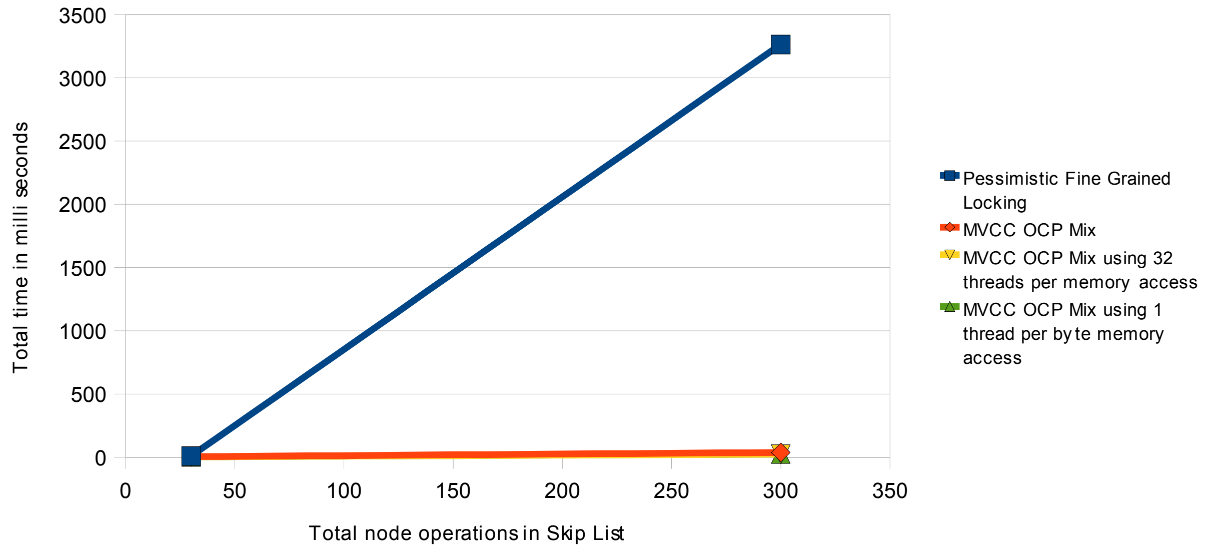

Fig.8 Performance graph for different concurrency controls for Skip-list

| **Total Nodes in Skip List** | **Total time in milli seconds using Pessimistic Fine-grained Locking** | **Total time in milli seconds using MVCC & OCP Mix using 1 thread per memory access** | **Total time in milli seconds using MVCC & OCP Mix using 32 threads per memory access** | **Total time in milli seconds using MVCC & OCP Mix using 1 thread per byte per memory access** |
|---:|---:|---:|---:|---:|
| 30 | 9.7 | 5.45 | 3.13 | 3.18 |
| 300 | 3264 | 46 | 25 | 26 |

Table 2 Performance in milli seconds for different concurrency controls for Skip-list

The above table and figure shows that with the growth of skip-list, the Pessimistic approach shows worst performance while the combination of MVCC and OCC shows a healthy performance gain. This performance is improved with the used of multi-threading in CUDA STM as explained in section 6.6.

Using of locks brought various considerations regarding the disadvantages discussed in the start. I decided to restart the transactions which comes in dead lock. I could reduce the blocking by using the fine grained locking. As explained in previous point, in a read operation, lock is not acquired now so read operation is non blocking.
At this point I had to decide about dealing with conflicts in concurrent transactions. Since I was using only one type of lock i.e; the write lock, therefore an obvious conflict is write-write concurrent access. In this situation, one of the writers should succeed in acquiring the lock while others may wait or move to acquire the next lock according to transaction log records. There are two possibilities in this case

1. If the conflicting transactions were working on same data items which they had first read

and now going to update, then if one of the transaction acquies al the locks and update the data items, it will after all update the associated version numbers as per MVCC approach. This will invalidate the read set or snapshot of data items in all other conflicting transactions. So there is no need for an unsuccessful transaction to move to try acquiring the next lock.

2. If acquiring of lock will not invalidate the read set of other concurrent transactions for example the lock was shared (section 6.2.1) for two different data items, in this case as well if an unsuccessful transaction had read the data item before and now trying to update it, still the successful transaction will update the version number associated with the lock. Thus invalidating the read set of any other transaction trying who tried acquire the same lock.

In both cases, the transaction which fails to acquire a lock will have no reason to continue and should be restarted.

In case of reader-writer conflict, in case the data item is under write lock, there is no read for reader to read because it may read corrupt data or invalid version number or both. On the other hand if the version number before and after read is not same, then reader can try again provided another transaction has not acquired the write-lock over that data item.

An important design decision was how much information should be maintained in transaction log. Traditionally transaction using direct updates [3] also have undo logs to revert the changes made by a transaction up-to the point of conflict. I already made the decision to make CUDA STM to perform deferred updates [3] therefore at-least the buffer and information regarding locks and versions should be maintained in transaction log . Transaction log is discussed latter in detail.

Final step of a transaction is called *commit* phase in which the CUDA STM acquires locks and verifies version numbers for validation of data and if validation and lock acquiring is successful, writes back the data to shared memory.

At last I arrived at complete design of CUDA STM. As per this design, this STM should use transaction local buffer to copy the values from global shared memory then perform computation on them and then write them back after acquiring the lock/s and validating the version number/s. CUDA STM uses an OCC & MVCC mix approach for concurrency control. It maintains a snapshot of data items by storing version numbers and then validates the read set with reference to these version number/s during commit. Only write locks are used  I had to decide which locks to acquire and which not in commit phase.

## 6.1 CUDA STM Architecture with respect to Databases

CUDA STM transactions ensure three basic properties of database transactions namely Atomicity, Consistency and Isolation or in abbreviation ACI.

1. *Atomicity* is either the full completion of a transaction or no effects of the transaction is ensured by acquiring all the locks of all the memory location to be written when the transaction is in commit phase and verifying version numbers of all the data items in read set. In this phase the status of all the locks is write-lock this prevents other transactions to accesses these objects and the current transaction can complete fully. In case any lock is not available or its version number does not match, then current transaction is aborted and restarted. This ensures proper atomicity.
2. *Consistency* is assured by read versioning of data structures. Consistancy refers to that only the valid data is to be written back. The version number of every data structure or, depending upon the implementation of kernel, even a char element of a data structure used in the transaction must not change till the commit phase. Since optimistic concurrency control is used in CUDA

STM, it allows acquiring of lock in write mode even if another transaction is reading that data item. In such a case, the reader transaction shall read again as explained before. Version verification and acquiring of locks ensures that the computed values written back to shared memory were based on most current values in shared memory.

3. *Isolation* is markable occurrence of every transaction in time line and is different for each transaction. In CUDA STM, rigorous testing was done on conflicting transactions and the architecture is robust enough to allow one transaction out of all the conflicting transactions to commit at one time instance. This increases the number of aborts with the increase in number of conflicting transactions but ensures the isolation property. The number of aborts increases with the increase in number of data items being accessed also. But only acquiring locks for write access although lowers the isolation but it also lowers the number of aborts as well. And with continuous testing, no data conflicts were found. Serializability is ensured by version number verification since all the version numbers related to data items written back are increased serially after very successful write.

## 6.2 CUDA STM Locking Mechanism

The concurrency can be increased to the level of a single member of the data structure, provided that a lock and a version number are associated with that data structure member. In such a case, the data structure should have a version number and a lock as well. This is implemented by separating the locking mechanism from data. The following data structure is used for locking and read versioning:

```
struct lockAndVersion
{
        int lock;
        unsigned int version;
        lockAndVersion()
        {lock = version = 0;}
};
```

*lock* means the lock. If its value is equal to 0, it means there is no writer active for the data element related to this lock and it can be acquired by a writer; or a reader can read if lock value is 0. To acquire the lock in write mode, lock value MUST be equal to 0 indicating there is no active writer and then it is set to -1 indicating a write lock. A write lock is not sharable. This prevents dirty reads. Read method does not read anything if the lock is in write mode. The lock acquiring in write mode fails if there is at-most one active writer.
The version is originally the number indicating number of times a particular data item was updated. To update a data item, the related lock is acquired in write mode which in other words means the version number can be associated to lock i.e; whenever the lock is acquired in write mode and is released only after the successful run of transaction, the version number should be increased. So both the lock and version number can be put in the same data structure.
Whenever a transaction tries to read a memory location, a lock is to be associated at that time along with version number. This is explained in section 6.2.1.
Memory address is stored in 4 bytes in C. This gave two options to implement the locks. Either to use a single 8 bytes unsigned long long data type and use its lower 4 bytes to store the lock value and upper 4 bytes to store the version number. This approach was implemented and tested. It made the transaction log easy to implement but was less flexible for code management because the lock part or

the lower 4 bytes are to be used as signed integer while the upper 4 bytes or the version number as unsigned integer and the pointer operations often confused me if I wanted to perform a minor update in the design of transaction log. The other approach is to use a separate 4 byte integer for lock and unsigned integer for version number. This made the architecture more flexible for further experiments because I could easily make updates in design and did not confuse the items. This approach was finalized. This helped simplifying the read versioning control of STM as well.

### 6.2.1 CUDA STM Shared Locking

Now I explain shared locking. While developing CUDA STM, it was noticed that as much locks are required as are the total data items. This is an excessive use of memory. Therefore another idea to share the locks was tested. According to this, a hash function is used which gives the lock index. Since the data items reside in global memory which is visible to every thread in any thread block, therefore locks are mapped on memory address and hash function gives same lock index related to a particular memory address in global memory. Thus lesser locks can be used and data items can share the locks. This has no effect on performance since a conflict is rare that same transaction tries to acquire same lock which it has already acquired for one memory address but according to output of hash function, same lock was allocated to another memory address which by chance is to be acquired in the same transaction. Even if this happens, the lock acquiring method checks it and continue for next data item in the log and prevents transaction from falling into an indefinite loop of trying to acquire an already acquired lock then abort and then again trying for the same lock. In case of binary tree, there was no problem of repetition of locks but in case of skip-list which requires a huge number of locks due to excessive updation of node pointers in insert phase, this problem was observed. Enumeration values were used to caution CUDA STM to check if a to-be acquired lock is already acquired by the same transaction or another transaction. Enumeration value "STM_DUPLICATE_LOCK_CHECK" is to be passed to Commit() function to make CUDA STM check for each non available lock. This slows down the lock acquiring process. Otherwise the default vale is "STM_NO_DUPLICATE_LOCK_CHECK".

### 6.2.2 CUDA STM Flexible Locking

CUDA STM provides highly flexible locking mechanism. The programmer can decide if the data item that is being read will be updated also or not or if there will be a lock acquired for that data item when the transaction is in commit phase. This is achieved by specifying a single character parameter for Read method. If a 'W' is provided then the STM will write back to this data item in commit phase. However in case of read only access where only the version number is to be verified then read with letter 'R'. For ease of use, an enumeration is used as parameter to Read() method which has two options "STMREAD" and "STMWRITE". A read access can be changed to write access during transaction but before commit phase. By default this value is "STMREAD".

## 6.3 Transaction Log

Each transaction must maintain a transaction log data structure which keeps meta data of the transaction. Despite the data type, every pointer stores memory address in 32 bytes. By using a 32 byte unsigned integer, the memory address can be stored and can latter be used to initialize a pointer by using reinterpret cast. This gives a total generalization for any type of data structures in CUDA STM through pointer operations. This is explained latter.

Transaction log contains an array of unsigned integers to store the lock address related to every data structure that a transaction reads or writes. Every transaction maintains a local copy of the data structure read and all the update are performed on this local copy and later written to the original

memory location depending upon various conditions explained latter.
There is another array in transaction log for storing the addresses of the version numbers of the data structure/s read or to be written during the transaction. To store the actual version number of the data structure there is a separate array. The version number is then read using the C pointer type casting of the address of version number stored in the array when transaction goes in commit phase. The same type casting is used to acquire the lock.
There is a separate array of acquired locks. When a lock is successfully acquired in commit phase, its address is stored in this array. This helps in identifying which locks the transaction has so far acquired and in case of abort, only these locks are released and not all the locks are to be checked because all the transactions are accessing the same locks and which transaction has acquired which locks is impossible to indicate at run time unless each transaction maintains its own information for regarding that. In case of abort, only these locks are released. This is a safe approach.
There is an abort counter integer that stores the number of times the transaction aborted. The starting and ending time of the transaction are also stored in the transaction log. This data structure turned out to be quite handy for storing the debug information during the test-run of the STM.
CUDA STM reads data byte by byte and copies it to transaction buffer inside transaction log. For that, a char data type pointer is sufficient which points to first byte of the shared data structure/s being accessed during the transaction and an additional information of data structure size fulfills CUDA STM 's generalization requirement. To implement this, three arrays are used, first one contains the pointers to shared data structures in global memory, a second one keeps their sizes and the third one keeps the pointers to local data structure clones in transaction buffer.
Transaction log or meta data is filled sequentially as the transaction proceeds. The main insertion point for meta data is Read method. So which ever data structure or data item is to be included in the transaction, it must be at least once read successfully to store all the necessary information about it. This helps in automatic updation of data in Commit phase and the programmer just has to specify the data items he or she wants to use by reading them and updating the buffer and then commit. The rest of the tasks are performed by CUDA STM. This has made an ease of use and simpler code to be written for kernels. Please read the section related to transaction buffer also.

## 6.4 Transaction Buffer

Every transaction is required to keep a local buffer. With respect to MVCC, a buffer can be considered as the snapshot of data in global memory. As transaction proceeds, its snapshot is developed till at the end, it is completed. Here, by snapshot I mean the versions of locks are being stored in the log and verified in commit phase.
When transaction reads a data structure or a data item, it gets an exact copy or clone as a local copy to perform operations upon. Well this should have been an ideal case except that in CUDA, although if a variable is declared in a function, we can pass its reference as pointer and access it anywhere else while that function or code region is still active, but when its memory address is stored in transaction log and is latter accessed in Write method, nothing is retrieved not even if the variable is declared in block level shared memory. This is quite a problematic situation because CUDA STM is supposed to simplify the kernel code as much as possible. It was observed that if the buffer variables are present in device global memory just like the original shared data structures, then buffer is accessible anywhere else latter on. It provided a remarkable simplification of kernel code. Pointers to buffer items are stored in transaction log when Read method is used. Whenever the global memory is accessed, the buffer is first checked to contain its clone. If a clone is found the pointer to clone is returned instead of reading from global memory. This reduces data contention and makes transaction faster. Latter on when data items are written back in commit phase, CUDA STM just checks the pointer to buffers and original shared data

structures in transaction log arrays explained before, the respective pointers are stored sequentially therefore there is no confusion in finding out a pointer to global shared memory and its respective size and buffer item related to it because all of them are at same index in different arrays.

## 6.5 Dynamic Memory Allocation for CUDA STM Components

Transaction log and locks are invisible components to the STM user. Throughout the kernel code, a developer has not to access them because they are automatically used and updated by CUDA STM. Each thread block uses a separate transaction log therefore as much transaction logs are required. Since the locks are shared as explained before therefore suitable number of locks have to used. The user has to decide about the maximum records in transaction log, maximum buffer size  and maximum number of locks that are to be used. The method *init()* accepts four parameters for number of blocks , maximum log entries, buffer length and; number of locks in the form of power of 2.

## 6.6 Multi-threading Support in CUDA STM for Memory Coalescing

CUDA STM provides support for multi-threading as well for copying data to and from global shared memory and buffer. Since data transfer is done byte by byte, therefore it  is a significant overhead  on STM performance. As explained before, a warp consists of 32 threads. Therefore a whole warp can be used for data transfer in which each thread transfers one byte at a time. Multi-threading support is present in internal STM functions but its up-to the user to get the benefit from it or just use single thread instead. Barrier synchronization is required in kernel code which can be done by using *__syncthreads()*. By using multi-threading in read and write, CUDA STM provides an excellent example of memory coalescing.

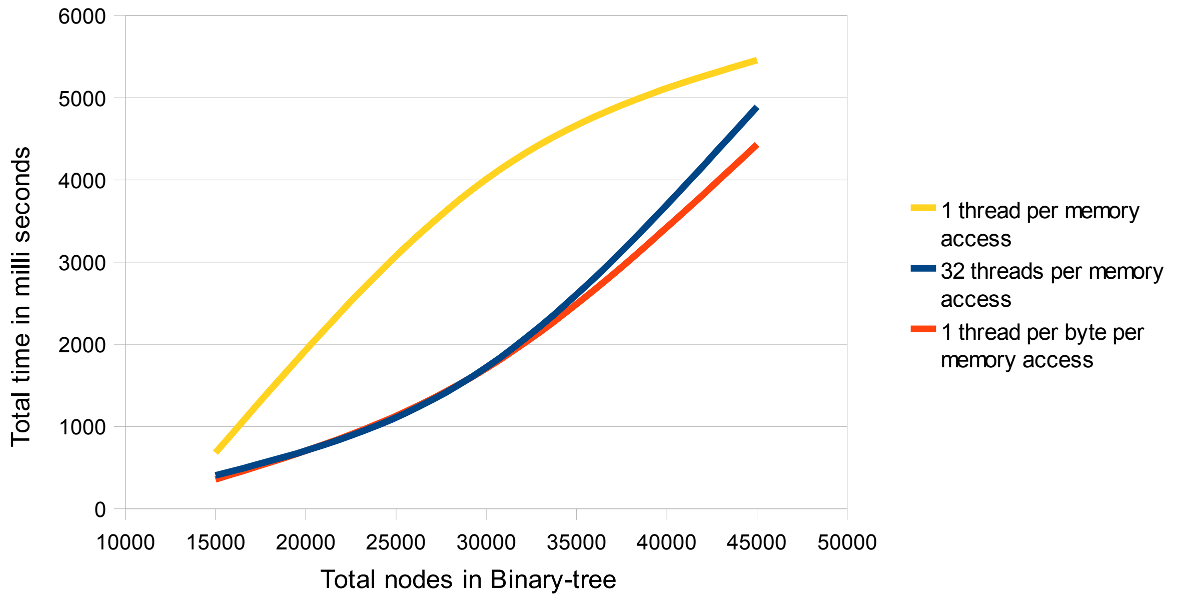

Fig.9  Maximum performance graph for different concurrency controls for Binary-tree

| **Total nodes in Binary-tree** | **1 thread per memory access** | **32 threads per memory access** | **1 thread per byte per memory access** |
|---:|---:|---:|---:|
| 15000 | 680 | 405 | 356 |
| 30000 | 4009 | 1719 | 1706 |
| 45000 | 5458 | 4889 | 4430 |

Table 3 Maximum performance in nodes and milliseconds for different concurrency controls for Binary-tree

The above figure and table show that using multi-threading for memory coalescing is beneficial for performance. Here one thing is notable that with the growth of binary-tree, the path from root to point of insertion of new node becomes longer and longer and versions of all the nodes on that path have to be verified before inserting the node. This takes much of the execution time and increases the number of aborts. But there is only one write-lock related to parent node of new node to-be inserted is to be acquired on whole path.

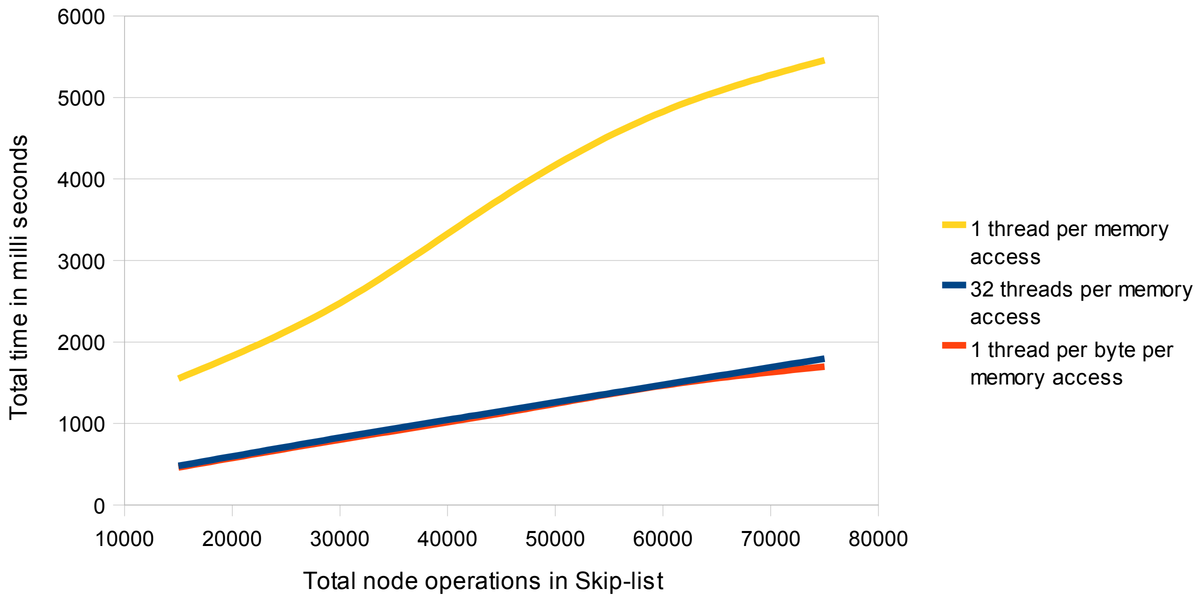

Fig.10 Maximum performance graph for different concurrency controls for Skip-list

| **Total node operations in Skip-list** | **1 thread per memory access** | **32 threads per memory access** | **1 thread per byte per memory access** |
|---:|---:|---:|---:|
| 15000 | 1550 | 478 | 460 |
| 30000 | 2476 | 825 | 800 |
| 45000 | 3767 | 1151 | 1125 |
| 60000 | 4827 | 1475 | 1465 |

|  |  |  |  |
|---:|---:|---:|---:|
| 75000 | 5458 | 1795 | 1697 |

Table 4  Maximum performance in nodes and milliseconds for different concurrency controls for Skip-list

Skip-list shows better performance than binary-tree because of its architecture. Skip list requires lesser number of jumps to reach appropriate node position. Skip list, as the name depicts, skips intermediate nodes while surfing. Thus it has smaller path from head to desired position. Skip-list provides more isolation to transactions because many transactions may be updating different parts of skip-list and are independent to each other. Therefore there are less chances of collisions or conflicts. Skip-list gives better response time than binary-tree due to these features.

## 6.7 CUDA STM Provides Alternative to Recursive Functions

As pointed out in section 5.4 that CUDA does not support recursive functions. But in CUDA STM, kernels can be created with proper logic that can perform the same tasks e.g; binary-tree and skip-list without using recursive calls that can work in concurrent environment.

Chapter 7

# How to write STM Transaction Kernel Code ?

Now I explain an example of code to use CUDA STM.

## 7.1 Example 1

Here is a simple example to explain the use of CUDA STM. There is a single data-structure that all thread blocks will update by a random value.

### 7.1.1 Make suitable data structure

```
struct test
{
        int value;
        test(){value = NULL;}

};
```

### 7.1.2 Declare Shared Variables

Shared variables are to be used if multi-threading is utilized. These variables are present in shared memory of a thread block and therefore are visible to all the threads in a block. Otherwise if they are not declared in shared memory but inside kernel, then each thread will have separate variables of its own and will disturb the synchronization.

```
__shared__  char *global;
__shared__  char *global_previous;
__shared__  char *temp_local;
__shared__  int value;
__shared__  int rval;
__shared__  test *tst_buffer;


__global__ void  Test_Kernel(node *p_test, int  blocks, int loop_run)
{



        int node_size = sizeof(test);
        rval = blockIdx.x * (blocks);


        for(int i = 0; i < loop_run; i++)

        {
```

```
                    if(threadIdx.x == 0)
                            {
                                    rval = ((rval * 214013L + 2531011L) >> 16) & 0x7fff;
                                    value = rval; // generate a random value
                       }

                __syncthreads();
        while(1)
                    {

                            Start();  //initialize the internal variables of  CUDA STM
                            result = 0;  //this is a global variable in CUDA STM for giving back responce of an operation
                            __syncthreads();
                            global = reinterpret_cast<char *>(p_test);
                            __syncthreads();
                            temp_local = Read(global, sizeof(test), STMWRITE);  //read shared memory
                            __syncthreads();
                            if ((result == 1)|| (temp_local == NULL)){   __syncthreads(); continue;} //check if the operation
                                                                                                //was successful otherwise restart the transaction
                           __syncthreads();
                            tst_buffer = reinterpret_cast<node *> (temp_local); //get the node read from buffer

                                    __syncthreads();
                                    if(threadIdx.x == 0)
                                    {
                                    nd_buffer->value = value; // set the root value
                                    }
                                  __syncthreads();
                                    Commit();
                                    __syncthreads();


                                    if(result == 0)
                                    {__syncthreads();break;}
                                    else
                                    { __syncthreads();
                                            Abort();
                                            __syncthreads();
                                            continue;
                                    }
                    }
            }
}
```

In the above *kernel* code, a single transaction runs in an indefinite *while* loop. This code is wrapped by a *for* loop which enables a thread block to perform more than one transactions. Methods like *Read* and *Commit*, update a shared variable *result*, which is 1 if operations generated an STM error. Otherwise it is 0. At anytime the transaction's *while* loop can be restarted. For that *Abort* method has to be called to clear the transaction log. *__syncthreads* is used at various places for barrier synchronization.

### 7.1.4 Allocate memory and call the kernel

The memory is allocated on device from the host and the original data is copied there from the host. Memory allocation is done as follows:

```
const int loop_run = 1; // If multiple transactions are required inside each block
const int blocks = 30;  // The number of blocks, each transaction is run on seperate block therefore it needs a seperate
// transaction descriptor, so this size is also the array size of transaction descriptors and
// for the convenience of the code, this is also used for the data structure array length
test  tst_host, *p_tst_device;  // one data structure instance and a pointer
```

Write the necessary functions to be executed on CPU inculing the *main()* function.

```
int main(int argc, char** argv)

{


        init(blocks, 5, sizeof(test), 0);  // 5 entries log and buffer size is enough to conatin one object since only one object
                                           //is being used for concurrent access in this simple example

        cudaMalloc((void**) &p_tst_device, sizeof(test)); //allocate suitable memory
        cudaMemcpy(p_tst_device, tst_host, sizeof(test), cudaMemcpyHostToDevice); //copy  data from host memory to
                                                                                  //graphics card

        Test_Kernel<<<blocks, sizeof(test), 0>>>(p_tst_device, blocks, loop_run);
          // number of threads is equal to the size of data-structure used in kernel for memory coalescing
        cudaError_t err = cudaThreadSynchronize();
        if(err != cudaSuccess)
                printf("Error: %s\n",cudaGetErrorString((cudaError_t)err));

      cudaMemcpy(tst_host, p_tst_device,sizeof(test), cudaMemcpyDeviceToHost); //copy data from graphics card to host
                                                                               //memory


        cudaFree(p_tst_device);  //free the memory on device


        printf("\n Node value is :  %d ",tst_host.value);//
        getchar();
        return;

}
```

There is a notable thing about calling the kernel from host code. Grid dimensions are determined by <<< >>> term. Its first parameter is the number of blocks in grid and second is the number of threads in each block. Threads in each block will be running the same kernel and so do all the threads in other blocks. However, the same number of threads will be provided to CUDA STM for its internal functions. Thus if there is only one thread in each block, then CUDA STM will read and write with one thread and there will not be a memory coalescing. If there are more than one threads in a block, then kernel programmer must control the threads so that all of them should together enter the *Read* and *Commit* methods. Once inside the STM internal functions, they will all come out together for sure due to CUDA STM design. To do so, *__syncthreads()* is a very useful utility to hold all the threads at one point in kernel code while one or more threads identified by their thread ID are doing any job.

## 7.2 More Complex Example

Here is a more complex example of a Binary-tree. This code was actually used to take performance results.

### 7.2.1 Make suitable data structure

```
struct node
{
        int value;
        node *left;
        node *right;
        int value_set;
        node(){value = NULL; value_set = -1; left = NULL; right = NULL;}

};
```

### 7.2.2 Declare Shared Variables

Shared variables are to be used if multithreading is utilized. These variables are visible to all the threads in a block. Otherwise if they are not declared in shared memory but inside kernel, then each thread will have separate variables of its own and will disturb the synchronization.

```
__shared__  char *global;
__shared__  char *global_previous;
__shared__  char *temp_local;
__shared__  int value;
__shared__  int rval;
__shared__  node *nd_buffer;
__shared__  node *my_nd_buffer;
```

### 7.2.3 Create the Kernel

```
__global__ void Generate_BinaryTree(node *root, node *l_nodes, int  blocks, int loop_run)
{



        int node_size = sizeof(node);
        rval = blockIdx.x * (blocks);


        for(int i = 0; i < loop_run; i++)

        {

                if(threadIdx.x == 0)
                        {
                                rval = ((rval * 214013L + 2531011L) >> 16) & 0x7fff;
                                value = rval; // generate a random value
                    }

         __syncthreads();
    while(1)
                {
```

```
Start();
result = 0;
 __syncthreads();
global = reinterpret_cast<char *>(root);
__syncthreads();
temp_local = Read(global, sizeof(node));  //read root node
__syncthreads();
if ((result == 1)|| (temp_local == NULL)){   __syncthreads(); continue;} //check if the operation
                                                                      //was successful otherwise restart the transaction
__syncthreads();
nd_buffer = reinterpret_cast<node *> (temp_local); //get the node read from buffer



if(nd_buffer->value_set == -1)  //if root was not set
{
        __syncthreads();
        if(threadIdx.x == 0)
        {
        nd_buffer->value = value; // set the root value
        nd_buffer->value_set = 1;
        nd_buffer->left = nd_buffer->right = NULL;
        Write_Intend(global); // the root will be updated
        }
       __syncthreads();
        Commit();
        __syncthreads();
}
else

        while(1)
        {
                __syncthreads();
                temp_local = NULL;
                if(value >= nd_buffer->value && nd_buffer->right != NULL) //navigate to
                 //right side of the current node if value is greater than or equal to node value
                {
                      global = (reinterpret_cast<char *>(nd_buffer->right));
                      __syncthreads();
                      temp_local = Read(global, sizeof(node));
                      __syncthreads();
                      if((result == 1)|| (temp_local == NULL)){__syncthreads(); break;}
                      __syncthreads();
                       nd_buffer = reinterpret_cast<node *> (temp_local);
                      __syncthreads();
                      continue;
                }
                else if(value < nd_buffer->value  && nd_buffer->left != NULL)//navigate to
                          //left side of the current node if value is less than the node value
                {
                        global = (reinterpret_cast<char *>(nd_buffer->left));
                       __syncthreads();
                       temp_local = Read(global, sizeof(node));
                       __syncthreads();
                       if((result == 1)||(temp_local == NULL)){__syncthreads(); break;}
                       __syncthreads();
                       nd_buffer = reinterpret_cast<node *> (temp_local);
                       __syncthreads();
```

```
                                        continue;
                                }


                                else {          //end of current path is reached after parcing from the root
                                        __syncthreads();
                                        Write_Intend(global);
                                        __syncthreads();
                                        global = (reinterpret_cast<char *>(&(l_nodes[blockIdx.x +
                                          (blocks*i)])));    //read the node related to current block
                                         __syncthreads();
                                        temp_local = Read(global, sizeof(node), STMWRITE);
                                        __syncthreads();
                                        if((result == 1)||(temp_local == NULL)){__syncthreads(); break;}
                                        __syncthreads();
                                         my_nd_buffer = reinterpret_cast<node *> (temp_local);
                                        my_nd_buffer->value = value;
                                        __syncthreads();
                                        if(value >= nd_buffer->value)
                                                nd_buffer->right = (&(l_nodes[blockIdx.x + (blocks*i)]));
                                        else if(value < nd_buffer->value)
                                                nd_buffer->left = (&(l_nodes[blockIdx.x + (blocks*i)]));
                                        __syncthreads();
                                        Commit();
                                        __syncthreads();
                                        break;
                                }
                        }


                        if(result == 0)
                        {__syncthreads();break;}
                        else
                        { __syncthreads();
                                Abort();
                                __syncthreads();
                                continue;
                        }
                }
        }
}
```

### 7.2.4 Allocate memory and call the kernel

The memory is allocated on device from the host and the original data is copied there from the host. Memory allocation is done as follows:

```
const int loop_run = 1; // If multiple transactions are required inside each block
const int blocks = 30;  // The number of blocks, each transaction is run on seperate block therefore it needs a seperate
// transaction descriptor, so this size is also the array size of transaction descriptors and
// for the convenience of the code, this is also used for the data structure array length
node nd_device[blocks * loop_run], *p_nd_device, nd_root, *p_nd_root;
```

Write the necessary functions to be executed on CPU inculing the *main()* function.

```
int validatetree(node* n)
{
        int c = 1;

        if(n->left!=0)
        {
                if(n->left->value > n->value)
                        printf("Invalid tree!\n");
                c+= validatetree(n->left);
        }

        if(n->right!=0)
        {
                if(n->right->value < n->value)
                        printf("Invalid tree!\n");
                c+= validatetree(n->right);
        }

        return c;

}

int main(int argc, char** argv)

{

        unsigned long long node_address_before = 0;
        unsigned long long node_address_after = 0;

        node_address_before = (unsigned long long) (&(nd_device[0]));
        init(blocks, 3000, 5000, 8);  // 3000 entries log and 5000 bytes long buffer

        cudaMalloc((void**) &p_nd_device, sizeof(node)*blocks*loop_run); //allocate suitable memory
        cudaMalloc((void**) &p_nd_root, sizeof(node));
        cudaMemcpy(p_nd_device, nd_device,sizeof(node)*blocks*loop_run, cudaMemcpyHostToDevice); /copy  data
from host memory to graphics card
        cudaMemcpy(p_nd_root, &nd_root,sizeof(node), cudaMemcpyHostToDevice);

        Generate_BinaryTree<<<blocks, sizeof(node), 0>>>(p_nd_root, p_nd_device, blocks, loop_run);
          // number of threads is equal to the size of data-structure used in kernel for memory coalescing
        cudaError_t err = cudaThreadSynchronize();
        if(err != cudaSuccess)
                printf("Error: %s\n",cudaGetErrorString((cudaError_t)err));


cudaMemcpy(nd_device,   p_nd_device,sizeof(node)*blocks*loop_run,   cudaMemcpyDeviceToHost);   //copy   data   from
graphics card to host memory
        cudaMemcpy(&nd_root, p_nd_root,sizeof(node), cudaMemcpyDeviceToHost);// the root node will show the
                                                                              //binary tree


        cudaFree(p_nd_device);  //free the memory on device
        cudaFree(p_nd_root);
```

```
    unsigned int address_difference = 0;
    address_difference = (unsigned int)p_nd_device-(unsigned int)&nd_device;//node_address_after - node_address_before;
            printf("\n Node 0 address before: %X",(unsigned int)p_nd_device);// node_address_before);
            printf("\n Node 0 address after: %X",(unsigned int)&nd_device);// node_address_after);
            printf("\n Node  address difference: %u", address_difference);
     if(nd_root.left != NULL)
            (nd_root.left) = reinterpret_cast<node *> ((unsigned long long) (nd_root.left) - address_difference);
            if(nd_root.right != NULL)
             (nd_root.right) = reinterpret_cast<node *> ((unsigned long long) (nd_root.right) - address_difference);
            for(int i = 0; i < blocks*loop_run; i++)
            {
                        if(nd_device[i].left != NULL)
                                    (nd_device[i].left) = reinterpret_cast<node *> ((unsigned long long) (nd_device[i].left) -
                                     address_difference);

                        if(nd_device[i].right != NULL)
                                    (nd_device[i].right) = reinterpret_cast<node *> ((unsigned long long) (nd_device[i].right) -
                        address_difference);
            }

            printf("\n After the tree......(Place debugger break point here and check \"nd_root\")\n\n\n");// place debugger at this
                                                                                                                         //line

             //Validate tree
            printf("Total items in tree: %d\n",validatetree(&nd_root));


            getchar();
            return;

}
```

In the host code, memory manipulation is required after the data is copied backed from graphics card because in a binary tree, a node uses pointers to point to next node and the memory address on graphics card and host memory are different.

# Chapter 8

## References

[1] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee 2008. “Software Transactional Memory: Why Is It Only a Research Toy?” Queue 6, 5, 46–58.
[2] Pascal Felber, Christof Fetzer, Rachid Guerraoui and Tim Harris, “Transactions are back—but are they the same? “Le Retour de Martin Guerre” (Sommersby)”” In ACM SIGACT News, March 2008 Vol. 39, No. 1 Pg 48 to 55
[3] James larus and Christos Kozyrakis, “Transactional memory: Is TM the answer for improving parallel programming?”, Communications of the ACM, Vol. 51, No. 7, July, 2008, Web science, ACM, New York, NY, USA
[4] Philip A. Bernstein and Vassos Hadzilacos and Nathan Goodman, “Concurrency Control and Recovery in Database Systems”, chapter 5, publisher Addison-Wesley,1987,
isbn 0-201-10715-5
[5] Philip A. Bernstein and Nathan Goodman, “Multiversion Concurrency Control - Theory and Algorithms”, ACM Trans. Database Syst., Vol. 8, No. 4, 1983, pg 465-483
[6] H. T. Kung and John T. Robinson, “On Optimistic Methods for Concurrency Control”, ACM Trans. Database Syst., Vol. 6, No. 2, 1981, pg 213-226, http://doi.acm.org/10.1145/319566.319567, db/journals/tods/KungR81.html
[7] Saha,, Bratin and Adl-Tabatabai,, Ali-Reza and Hudson,, Richard L. and Minh,, Chi Cao and Hertzberg,, Benjamin, “McRT-STM: a high performance software transactional memory system for a multi-core runtime”, PoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, 2006, isbn = 1-59593-189-9, pg 187-197, New York, New York, USA
[8] Torvald Riegel and Becker de Brum, Diogo, TRANSACT 2008, “Making Object-Based STM Practical in Unmanaged Environments”, February 23, 2008, 3rd ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, Utah, USA
[9] Adl-Tabatabai,, Ali-Reza and Lewis,, Brian T. and Menon,, Vijay and Murphy,, Brian R. and Saha,, Bratin and Shpeisman,, Tatiana, “Compiler and runtime support for efficient software transactional memory”, PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, 2006, isbn = 1-59593-320-4, pg 26 – 37, Ottawa, Ontario, Canada, ACM, New York, NY, USA
[10] M. Herlihy and V. Luchangco and M. Moir and W.N. Scherer, “Software Transactional Memory for Dynamic-sized Data Structures”, Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, July 2003
[11] NVIDIA® CUDA™ Architecture by NVIDIA Corporation 2701 San Tomas Expressway Santa Clara, CA 95050, http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf
[12] Herlihy, Maurice and Moss, J. Eliot B., “Transactional memory: architectural support for lock-free data structures”, ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture, 1993, isbn 0-8186-3810-9, pg 289—300, San Diego, California, United States; ACM,

New York, NY, USA
[13] NVIDIA_CUDA_Programming_Guide_2.1, version 2.1 dated: 12/8/2008 by NVIDIA Corporation 2701 San Tomas Expressway Santa Clara, CA 95050, www.nvidia.com
[14] http://en.wikipedia.org/wiki/CUDA
[15] "Concurrency control" by IBM, http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/cejb_cncr.html