Efficient Transactional-Memory-based Implementation of Morph Algorithms on GPU

Shayan Manoochehri

A Thesis
in
The Department of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Shayan Manoochehri

Entitled: Efficient Transactional-Memory-based Implementation of Morph Algorithms on GPU

and submitted in partial fulfillment of the requirements for the degree of

M. Comp. Sc.

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________________ Chair
Dr. T.-H. Chen

______________________________________ Examiner
Dr. H. Harutyunyan

______________________________________ Examiner
Dr. R. Jayakumar

______________________________________ Supervisor
Dr. D. Goswami

Approved by ______________________________________
Chair of Department or Graduate Program Director

____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Date ______________________________________

Abstract

Efficient Transactional-Memory-based Implementation of Morph Algorithms on GPU

Shayan Manoochehri

General Purpose GPUs (GPGPUs) are ideal platforms for parallel execution of applications with regular shared memory access patterns. However, majority of real world multithreaded applications require access to shared memory with irregular patterns. The morph algorithms, which arise in many real world applications, change their graph data structures in unpredictable ways, thus, leading to irregular access patterns to shared data. Such irregularity makes morph algorithms more challenging to be implemented on GPUs which favor regularity. The Borouvka's algorithm for calculating Minimum Spanning Forest (MSF), and multilevel graph partitioning are two examples of morph algorithms with varied levels of expressed parallelism. In this work we show that a transactional-memory-based design and implementation of the morph algorithms on GPUs can handle some of the challenges arising due to irregularities such as complexity of code and overhead of synchronization. First, we identify the major phases of the algorithm which requires synchronization of the shared data. If the algorithm exhibits certain algebraic properties (e.g., monotonicity, idempotency, associativity), we can use lock-free synchronizations for performance; otherwise we utilize a Software Transactional Memory (STM) based synchronization method. Experimental results show that our GPU-based implementation of Borouvka's algorithm outperforms both the fastest sequential implementation and the existing STM-based implementation on multicore CPUs when tested on large-scale graphs with diverse densities. Moreover, to show the applicability of our approach to other morph algorithms, we do a pen-and-paper implementation and complexity analysis of multilevel graph partitioning.

Dedication

I dedicate this thesis to my mother and brothers who always supported me with love, and to the memory of my father who taught me to be strong and kind to others.

Table of Contents

List of Tables, Figures and Algorithms

1 INTRODUCTION

An application with data-level parallelism (DLP) contains computations that can be performed on many data items simultaneously. In contrast, the task-level parallelism is expressed in computations wherein multiple tasks can be executed in parallel. An application with data-level parallelism scales with the increase in amount of data, whereas an application with task-level parallelism scales as the number of functionalities increase [1]. GPUs are throughput-driven, highly multithreaded processors benefiting the applications with large data-level parallelism to achieve high performance. DLP computations can either be performed in regular or irregular fashion, as we discuss in the following.

An application's behavior, with respect to the control flow and memory access patterns, is largely determined by its degree of dependency to the values of data items. When the dependency is predictable or follows a pattern, the data-parallelism is regular; otherwise, the behavior of the application is irregular. In regular parallelism, roughly identical computation is performed on each data item; therefore the load balance on processing cores is easy to achieve leading to efficient utilization of the system resources. Furthermore, the predictable memory access patterns can be exploited to improve the efficiency of memory transfers, i.e., memory accesses exhibit spatial and temporal locality as the result of predictable behavior expressed in the regular computation which can be exploited for DLP computation optimizations.

On the contrary, in irregular parallelism, the amount of computation performed in each data item can be quite variable. The challenges to efficiently perform these computations are mainly due to: (1) the control flow of each data item's processing is data-dependent, which leads to load imbalance. (2) The branch divergence arising due to if-then-else constructs, which can lead to underutilization of the SIMD hardware in GPUs. (3) The irregular memory access patterns, which make it difficult to group memory transfers that exhibit temporal and/or spatial locality into one memory request (memory coalescing), leading to underutilization of the high memory bandwidth available in GPU.

Widespread deployment of GPUs in the recent years has resulted in substantial speedup of applications with regular memory access patterns. However, many real world applications exhibit computations with irregular DLP. Some of the irregular computations are originated from the complexity of data structures they use, e.g. graphs. Morph algorithms [2] are a type of irregular algorithms that change the underlying graph data structure in unpredictable ways, making them difficult to exploit memory coalescing and mitigate control flow divergence on GPUs. Moreover, the implementation of the morph algorithms in massively multi-threaded environments give rise to synchronization problems caused by dynamic data sharing. This leaves us with a question with no clear answer: how should we reduce the cost and complexity of synchronization overheads in these applications to be able to fully utilize the processing power of the thousands of GPU cores?

Examples of some of the morph algorithms are: Borouvka's Minimum Spanning Tree (MST) algorithm [3] to find a minimally weighted subset of the graph edges that connects all the vertices without any cycle; multilevel graph partitioning to divide the underlying graph to partitions of roughly equal size via three phases: coarsening, initial partitioning, and un-coarsening; and Delaunay Mesh Refinement (DMR) to transform triangles of the given triangulated input mesh that do not conform to certain quality constraints by recreating the neighborhood around them. Parallelizing morph algorithms on GPUs exemplifies the challenges of executing irregular real world applications, and is the focus of our research.

Over the past few years, there have been some implementations of morph algorithms on GPU, which are relatively complex and non-scalable. Vineet et al [4] implemented a recursive formulation of Borouvka’s algorithm on GPU using CUDA primitives such as scan, segmented scan, and split. However, their approach is not only complicated but also doesn’t scale well to larger graphs due to the restrictions on the input graph. Nasre et al. [5] implemented the Borouvka’s algorithm on GPU, avoiding any synchronization all together. On the downside, they proposed a rather complex algorithmic technique deviating from the phases of the original algorithm, e.g., it included a phase to break the formation of cycles among components to prevent race conditions. Moreover, their approach did not prove to be quite efficient, which could be due to the extra work required to avoid synchronization, as we show in our experimental results.

On the other hand, there are other works which are not quite reliable, e.g., Nasre et al. [6] have proposed a techniques to improve the performance of DMR algorithm by performing a three-phase race-and-resolve synchronization method using global barriers. Unfortunately, global barriers can result in poor performance on GPU due to load imbalances caused by dissimilar workloads. Moreover, it has the potential of causing livelocks which makes it an unreliable technique.

Software Transactional Memory (STM) [7] is a software method supporting transactional programming of synchronization operations. It simplifies development of parallel code by allowing the programmer to mark sections of the code that should be executed concurrently and atomically in an optimistic manner. High abstraction level of STM makes the transition from the algorithmic design to implementation a natural process as opposed to tedious programming efforts required in a lock-based implementation.

Cederman et al. [8] implemented two versions of blocking and non-blocking STMs on GPUs that work with the granularity of one thread-block, which limit the proper utilization of system resources. They employed linear and exponential backoff functions to recover from transactional conflicts as well as avoiding livelocks. Later, Xu et al. introduced GPU-STM [9] with a hierarchical validation approach which is a combination of timestamp-based and value-based validations. They used locks-sorting to avoid livelocks. None of the above STMs showed good scalability, mainly due to expensive conflict detection mechanism they employed.

Following Xu et al. [9], Holey et al. [10] presented a scalable Lightweight Software Transactions for GPUs with a back-off strategy to avoid livelocks. Their work is more efficient as compared to GPU-STM as it lowered the overhead of STM by introducing two variations of STM supports: ISTM (invisible reads) and PSTM (pessimistic). ISTM introduced invisible reads to reduce conflicts during a transaction where reads with no modifications to memory locations comprised the good portion of memory accesses. PSTM is a simpler, yet more effective, method in majority of applications where transactions regularly read and write to the same memory location. It treats the writes and reads in the same manner, leading to simpler design and reduced overhead, therefore, resulting in higher scalability especially when the contention level is high.

Shen et al. [11] introduced a priority rule based STM (PR-STM) which incorporates a lock stealing technique to avoid the possibilities of both livelocks and deadlocks by prioritizing transactions based on the threads' identifiers. We built our STM on top of PR-STM and incorporated the optimization technique used in PSTM to lower some of its overheads. Both Shen's and Holey's work show a good level of scalability in the benchmark applications they used. However, we chose to build on the work of Shen mainly because of the backoff strategy which is not a natural match for GPU due to its SIMD hardware.

In this thesis, we demonstrate efficient transactional-memory-based design and implementation of some of the morph algorithms on GPUs. We believe that transactional-memory-based design facilitates the scalable implementation of the algorithm and reduces the amount of programming effort. In each major phase of the algorithm, we first attempt to extract certain algebraic properties of the computation (e.g., monotonicity, idempotency, and associativity) [6] to exploit lock-free transactions for performance. Otherwise, we utilize an STM based synchronization method. We also apply different optimization techniques like "*warp segmentation*" [12] in our implementation to maximize the load balance which can lead to proper GPU utilization and good performance.

To show the effectiveness of our transactional-memory-based approach, we implemented the Borouvka's algorithm on GPU: in the first major phase of the Borouvka's algorithm (edge discovery), we extract certain algebraic properties of the computation (monotonicity) [6] to exploit lock free synchronization. The edge discovery phase is monotonic due to ever decreasing values produced in finding the MSF edges. However, due to lack of such properties in the merge phase, we apply STM based synchronization, using our priority-based STM implementation. Moreover, to show the applicability of our approach to other irregular algorithms, we perform a pen-and-paper design and analysis of a transaction-memory-based multilevel graph partitioning algorithm, wherein STM transactions are used during coarsening (matching stage) and un-coarsening (refinement stage) phases.

The main contributions of this research are as follows:

- Design and implementation of an STM-based Borouvka's MSF algorithm. The STM-based approach improves the programmability and cost of synchronization as compared to some other existing approaches that use fine-grained locks or algorithmic methods to avoid locks altogether. The STM-based approach is also simple to implement, which reduces the gap between the original algorithm and its actual implementation.

- Implementation of a priority-based STM to be used in the synchronization steps of a morph algorithm.
- Comparison of our experimental evaluations with the sequential and STM-based multicore implementations of MSF for large graphs (both real-world and artificial) demonstrates substantial performance improvement.
- We demonstrate the applicability of our approach to other morph algorithms by discussing a pen-and-paper design and complexity analysis of the multilevel graph partitioning algorithm.

The rest of this thesis is organized as follows: chapter 2 elaborates the background information including GPU architecture, Transactional Memory, and an overview of STM implementations on GPU. Chapter 3 elaborates the design and implementation of the transactional-memory-based Borouvka's algorithm on GPU and our priority-based STM; this is followed by a discussion on the experimental evaluations and analysis. In chapter 4 we discuss the design and analysis of a transactional-memory-based multilevel graph partitioning algorithm. Finally, Chapter 5 concludes the thesis with a discussion on future works.

2 BACKGROUND

2.1 GPU Architecture

In this section, we first start by outlining the system model of GPU and then explain GPU's synchronization mechanisms, namely atomic operations and barriers. Finally, the challenges of using locks in GPU are discussed.

2.1.1 GPU System Model

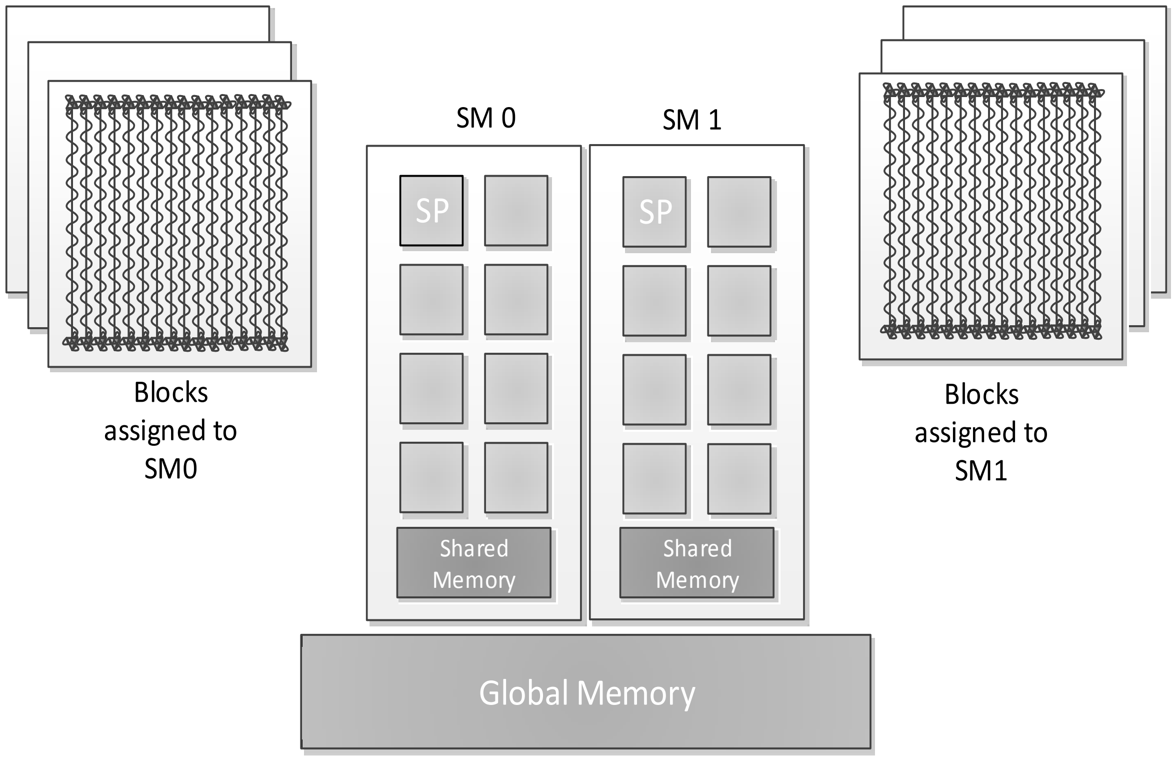


Figure 2.1. GPU's Execution Model.

A GPU application starts on the CPU and uses a compute acceleration API such as CUDA [13] to launch a highly multithreaded kernel onto the GPU. Introduced by NVIDIA, CUDA is a general purpose parallel computing architecture to execute compute-intensive programs on NVIDIA's graphics processors. Moreover, CUDA provides a compiler for a language based on C/C++ with the extensions in order to run kernel codes (functions) on the NVIDIA's GPU. In the following we use all the terminologies related to CUDA.

GPUs support thousands of threads executing in parallel. Such computing power is provided by an array of streaming multiprocessors (SM). Each kernel launch consists of a hierarchy of threads, called a grid, executing the same compute kernel. The thread hierarchy organizes threads as thread blocks wherein each thread block is scheduled to one of the SMs as a single unit of work, staying there until all of its threads complete execution. At execution time, threads within the same thread block are further partitioned into warps [13] to exploit their regularities and spatial localities.

As it is shown in Figure 2.1, each SM consists of an array of simple cores that are referred to as streaming processors (SP). In a given cycle, the SPs belonging to a single SM execute the same instruction but on different memory locations, an execution model known as single instruction multiple data (SIMD) or single instruction multiple threads (SIMT) [13] due to hardware threading. Threads within the same warp are simultaneously scheduled to the SIMT cores or SPs, and thus are executed in lockstep fashion, while threads across different warps are executed asynchronously in a multiple instruction multiple data (MIMD) like fashion. Moreover, using memory access coalescing mechanism, multiple consecutive accesses to the same DRAM row are coalesced to generate a single memory request, leading to an efficient memory operation.

Moreover, the arrangement of threads into warps by the programmer provides optimizations with respect to control flow. Ideally, threads within the same warp execute through the same instruction path, so that the GPU's SIMT cores can execute them in lockstep fashion. However, a warp may encounter a branch divergence when its threads diverge as the result of data-dependent branches. Modern GPUs contain masking hardware to handle branch divergence until the threads re-converge.

Note that GPU kernel can be launched with any number of thread blocks that can go beyond the capacity of GPU on-chip resources. The thread blocks of a grid are enumerated and distributed to multiprocessors with their available execution capacity, i.e., the launch unit dispatches as many thread blocks as the GPU's capacity, and dispatches the remaining thread blocks when the resources are released by completed thread blocks; this process ends when all the thread blocks are dispatched and completed. Such hardware-accelerated thread scheduling mechanism distinguishes GPUs from other processors such multi-core. Therefore, given such mechanism, GPU applications can decompose their workloads into as many threads as possible without imposing any major overhead.

The GPU contains an on-chip memory in each SM, which is explicitly managed by a programmer, referred to as the shared memory. The shared memory is accessible among different threads of each SM-assigned thread block. On the other hand, global memory, which is an off-chip device memory, is accessible to all threads in a GPU kernel. The access latency of device memory is high which can potentially make the memory accesses costly.

To compensate for such high memory latency, (1) consecutive memory accesses to device memory from different threads in a warp are automatically coalesced, i.e., combined into a single larger access, exploiting GPU's large memory bandwidth, and (2) fast context switching in each processing core help to perform some useful computation while waiting for the memory operation to complete, where only a limited number of threads are allowed on each processing core. As opposed to global memory, local memory is private to individual threads, although, it also resides in the off-chip device memory.

2.1.2 GPU's Synchronization Mechanisms

Atomic Operations

GPUs provides hardware-based atomic operations for the GPU kernels to synchronize critical sections with supports for both shared and global memory spaces. Such atomic mechanisms are capable of updating a memory location atomically with a new value computed based on the old value. An atomic operation involves 3 steps: reading the old value from the memory location, modifying it into a new value, and writing the new value back to memory where all three steps are performed atomically with respect to the thread executing the atomic operation.

Moreover, the atomic operations are the basic primitives for constructing synchronization mechanisms such as locks, barriers, non-blocking data structures and software transactional memory. Among different types of atomic primitives, atomic compare-and-swap (CAS) is the basis to build other atomic primitives in GPUs such as atomic min, max, add and etc. An atomic CAS receives 3 input parameters: the memory location to be updated, the expected old value at the location, and a new value; this primitive will write the new value into the memory location only if the location's current value is equal to the expected old value. If the operation is successful the old value is returned to the thread, otherwise, the new value is returned to indicate the failure, i.e., the atomic primitive of current thread aborts as other thread has changed the value of the location before the current thread has a chance to complete. The CAS may be repeated using a loop construct until it succeeds.

Barriers

A barrier provides synchronization among a set of threads such that none get past the barrier until every thread in the set has reached to the barrier. Barriers can be used in a programming method known as *bulk-synchronous programming* [14] wherein the computation is divided into multiple phases, with each phase consisting of a set of independent tasks executed concurrently, therefore, avoiding any race conditions. I.e., the barrier between subsequent phases ensures that all tasks are finished and their updates have reached to the global memory before proceeding to the next phase.

However, if the tasks are dissimilar, this process leads to load imbalance and underutilization of system resources. This is because the threads which have finished their assigned tasks quicker may have long idle time at the barrier, waiting for other threads to catch up. Therefore, bulk-synchronous programming is more effective for applications with regular parallelism, in which threads are assigned with roughly the same amount of workloads.

Multi-core processors usually implement barriers via the use of atomic operations. In the simplest implementation, each thread atomically increments a single counter, and then spins until the counter equals to the total number of threads participating in the barrier. Modern GPUs provide hardware barrier instructions that synchronize threads at the level of thread block (using __*syncthreads* in CUDA), which can be utilized to implement software-based global barriers which involves all the kernel's threads.

2.1.3 The Locks Challenges

In addition to traditional challenges of lock-based synchronization, concurrency issue can be even more troublesome due to GPU's SIMT execution paradigm. For example, as shown Algorithm 2.1, assume that there are two threads within a warp competing for a spinlock. In this situation, one of them acquires the lock, and waits at the start of critical section for re-convergence, while the other spins forever, leading to deadlock.

```
repeat locked = CAS(&lock,0,1)
until locked = 0;
//critical section …
Lock = 0;
```

Algorithm 2.1. Illustration of deadlock case in GPU.

```
done = false;
while done == false do
  if CAS(&lock,0,1) == 0 then
     //critical section …
     lock = 0;
     Done = true;
```

Algorithm 2.2. Illustration of simple fix to deadlock issue in GPU.

The updated version of Algorithm 2.1 (Algorithm 2.2) a local "boolean" variable is added to fix the issue. This algorithms works well when each thread acquires a single lock, however, when two threads from the same warp attempting to acquire two locks in reverse orders, it leads to livelock. I.e., when a thread incurs locking failure, it releases the lock acquired and retries but since warps execute the same instruction in lockstep fashion, those two threads loop forever. Therefore, due to livelocking issue, fine-grained locking on GPUs is extremely challenging.

2.2 Transactional Memory Model

In this section we discuss the basic idea and benefits of using Transactional Memory as opposed to conventional synchronization method by means of locks. Subsequently we discuss the progress guarantee provided by different types of systems underlying the Transactional Memory, and finally we discuss the common steps of transactional execution of a typical STM system.

2.2.1 Transactional Memory

Herlihy et al. [15] introduced transactional memory (TM), a new multiprocessor architecture implemented by straightforward extensions to any multiprocessor cache-coherence protocol, which provides an efficient and easy-to-use synchronization technique without using locks. In contrast, conventional mutual exclusion using locks ensures that only one thread at a time can operate within the critical sections, while serializing access to the shared memory locations. TM, on the other hand, can circumvent common problems associated with locks, including priority inversion, which occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads; convoying, which occurs when a thread holding a lock is de-scheduled causing other threads capable of running wait; and difficulty of avoiding deadlock.

TM allows programmers to define transaction as a block of code with read-modify-write operations which atomically updates multiple memory locations. Building on the hardware based TM [15], Shavit et al. [7] proposed a software transactional memory (STM), a software method for supporting flexible programming of transactions; their design, however, has a major practical limitation in its transaction definition which its data set and operations should be known in advance, i.e., it uses static transactions. This limitation was later addressed in the following TM designs, making the transactions dynamic.

The underlying TM system should ensure the transactions, are (1) executed atomically, meaning that either all or none of a transaction's operations are executed, and (2) in isolation, meaning that transactions do not observe un-committed updates of other transactions in (shared) memory. Therefore, the TM system can provide an abstraction which frees the programmer from the complexity and caveats of dealing with low level synchronization primitives. I.e., the TM paradigm greatly simplifies concurrent programming compared to the traditional lock-based techniques, that is, using TM system, the programmer does not need to write code via locks to ensure mutual exclusion.

The TM system automatically executes transactions in parallel for performance, while resolving data-races among concurrently executed transactions. The programmer only need to mark the code block which has to be executed atomically and the TM ensures its correct execution and good performance. With a well-implemented TM system, transactions can approach a performance comparable to that of fine-grained locking. Despite having a diverse programming syntax, TM code has been shown to be less error-prone than locks [16], while being easier to understand [17]. Moreover, the programmer can compose transactions using TM [15], which further boosts programmer productivity as compared to traditional programming using locks.

Even though a TM system execute transactions concurrently, to execute correctly it should satisfy the *serializability* as its basic correctness criterion. Serializability defines how a set of committed transactions may update the memory system [15], and it is satisfied when transactions appear to execute serially, that is, the steps of one transaction never appear to be interleaved with the steps of other transactions. Hence, The TM system should ensure that the current execution is serializable, that is, equivalent to executing the same set of transactions in a serial order. In a serializable order, every transaction behaves as if it is executed serially one after another. By showing that all transactions committed by a TM system satisfy serializability, atomicity and isolation are also satisfied. Note that serializability only requires the concurrent execution of transactions to have at least one equivalent serial order, that is, it does not restrict which serial order should be matched by the concurrent execution.

2.2.2 Progress Guarantees

Conventional lock-based concurrency control methods guarantee data consistency by enforcing mutually exclusive access to critical sections guarded by locks that are exclusively acquired by concurrent threads. Once a lock has been acquired by a thread, other concurrent threads demanding to acquire the same lock are forced into a wait state until the lock is released by the thread owning it. Therefore, the waiting threads may spin, yield, or ask the scheduler to block them, that is, they can't make forward progress until the lock is released. This wait state is the major cause of the various problems including deadlock, priority inversion and convoying.

Similar to the initial STM implementation [7], the proposed non-blocking STMs provide synchronization mechanism to circumvent the problems caused by mutual-exclusion using locks, so the failure of any number of threads could not prevent the remainder of the system from making progress. As a result, non-blocking STMs are not relying on the poor scheduling decisions and performed well in spite of arbitrary thread termination. To support non-blocking property, the non-blocking STMs exclude the use of locks because the locks could be held by threads and never be released. However, due to the efficiency limitations of the non-blocking STMs, most of the recent STMs are blocking, and hence, lock-based, an approach which has shown better performance in practice, partly due to its design simplicity.

In the non-blocking STMs, threads do not need to wait, e.g. spin, to gain access to a shared memory location in face of contention. That is, the transactions in a non-blocking STM are free from wait state, when run by concurrent threads. Instead of waiting, a thread may either abort and optionally retry later; or abort the conflicting thread. In other words, non-blocking STM permit concurrent reads/writes to the shared memory locations, while guaranteeing consistent updates using atomic primitives such as CAS. As we discuss in the following, the non-blocking STMs can be classified according to the kind of progress guarantee they provide to wait-freedom, lock-freedom and obstruction-freedom.

The strongest progress guarantee is wait-freedom which, while having the poor performance, guarantees a transaction always succeeds in a finite number of its own steps, therefore, eliminates the occurrence of deadlocks as well as starvation. This guarantee is usually only relevant to real-time systems, where predictability is compromised with performance.

A weaker and more practical guarantee is lock-freedom, which guarantees that, given a set of concurrent transactions, at least one transaction makes progress in a finite number of execution time steps of any other transaction. In spite of the name, lock-freedom does not necessarily preclude locks, as long as these can be revoked. Lock-freedom eliminates the occurrence of deadlocks but not starvation. E.g., the original proposal for STM suggested by Shavit et al. [7] was a lock-free STM with no use of locks.

An even weaker guarantee is obstruction-freedom. It guarantees that a transaction will always succeed if it is executed without conflicts with other transactions. This leads to greater simplification and flexibility in the design of non-blocking STMs. This property strong enough to prevent deadlock and priority inversion, but requires separate mechanisms such as exponential back-off to avoid livelock. A contention manager is often required to achieve high throughput and to prevent livelocks. Thus, in practice, obstruction-free STMs seem to be faster than wait-free and lock-free alternatives.

In contrast to the above mentioned progress guarantees the non-blocking STMs provide, the blocking STMs provide no guarantees at all since they use locks. Regardless of the lack of progress guarantees, they allow for a simpler design and, according to Ennals [18], a potentially more efficient implementation. Moreover, (1) the majority of programs incorporating STM use it to improve the performance, (2) in the programming models, it is acceptable for one atomic operation to block other atomic operations of the same or lower priority, and (3) the number of threads in an application can be adaptively be matched to the number of physical processing cores, thus, making it unlikely for a thread's transaction to be blocked by another transaction which its thread has been swapped out. The disadvantage of using a blocking implementation is that it makes the STM much more dependent on the scheduler.

2.2.3 Transactional Execution

In this section, we outline the common steps of transactional execution for a typical STM system. We start by giving a formal definition of transaction: A transaction is a sequence of instructions, including reads and writes to memory that either commits, that is, executes to the end atomically, or aborts with no effect. All the speculative writes to memory location performed by a committed transaction become permanent, and hence, visible to the rest of the system, while the speculative writes of an aborted transaction is discarded. The STM system usually needs a version management mechanism to record the speculative writes to ensure a consistent memory view from different threads, along with conflict detection mechanism to detect the dependence violations between different transactions.

Version management can be performed eagerly at the time of memory access, or lazily when the transaction commits. In the eager mechanism, the changes are optimistically applied directly to the memory locations as a transaction makes progress, while logging the old values stored in the memory locations to an undo log which is used to revert the changes in case of abort. Generally, eager version management performs well when conflict detection rate is low, while lazy version management is more effective in high contention workloads [19]. However, in the lazy, rather pessimistic, mechanism, the speculative writes will be logged in transaction-private redo logs until the commit time which is the time that the transaction's speculative writes are stored to memory, whereas the redo log is discarded when the transaction aborts.

Similar to version management, conflict detection can be performed either eagerly or lazily. Eager conflict detection detects violation early, thus, is able to lessen redundant work by the threads and make computation and memory resources available for other threads. Besides, with the eager conflict detection, the aborted transaction are allowed to retry early. On the other hand, in the lazy mechanism, the conflict detection is optimistically executed at the commit time, allowing for higher degree of concurrency due to the reduced abort rate. Note that the selection of conflict detection mechanism depends on the chosen version management mechanism. In eager version management, conflicts on writes need to be detected eagerly before any write to the memory locations, whereas the conflicts on reads can be detected either eagerly or lazily.

The data conflicts can be detected by tracking the transactional reads and writes to memory locations recorded by transactions within the (shared) memory, hence, making them visible to other transactions. To do so, most of the STM implementations dedicate a separate (shared) memory space for either one memory location or a group of memory locations (memory stripe) to record the tracking entries which include information such as version number, which is used for invisible reads, ownership (lock) status, current executing thread, operation type (read or write) and old/new value for each memory location accessed. Each transaction also keeps a local copy of such entries for each speculative read or write operation performed.

A STM system can abort transactions either explicitly based on the logic of code, or implicitly due to the conflicts with concurrent transactions. The implicit conflict detection along with resolution, which is the abort followed by retry, mechanisms avoid the deadlocks which naive fine-grained locking implementations suffer from. In practice, STM usually aborts transactions even before a deadlock takes place, typically as soon as the conflict is detected.

In a typical STM system, performing an *abort* operation, generally reverts all the changes made by the transaction, effectively voiding the entire transaction, and subsequently releases the ownership of the acquired memory locations. This is commonly referred to as transaction rollback. While when a transaction completed successfully via the *commit* operation, the changes to memory locations become visible to other threads by releasing the ownership of the acquired memory locations.

To maximize the degree of concurrency, STM implementations typically differentiate among reads and writes to memory locations, allowing several transactions to read the same memory location concurrently, as long as no update is performed by any transaction at the same time. This doesn't violate the data consistency as the transactions are only reading from the same location. On the other hand, the conflict occurs when at least one of the transactions writes to the memory location while any other transaction read from the same location.

2.3 Overview of GPU STMs

Providing efficient STM support on GPU is challenging due to sheer large number of lightweight threads in GPU as compared to few powerful threads in CPU. Such lightweight threads could experience substantial performance overheads with transaction management tasks such as tracking dependences (version management), detecting dependence violations (conflict detection), locally buffering data before a transaction commits, and re-executing aborted transactions across thousands of GPU threads at runtime.

In the following we briefly highlight some of the major works to design and implement STMs on GPUs and compare their effectiveness with respect to efficiency and scalability.

2.3.1 Block Level STM

Cederman et al. [8] have designed and implemented two versions of blocking and non-blocking STMs on GPUs that work with the granularity of one thread-block, thus, avoiding intra warp livelock. However, such granularity limits the proper utilization of system resources. Their non-blocking STM follows the obstruction-free design proposed by Harris et al. [20]. Both STMs perform lazy version management using the redo logs for best performance in applications with high contention, while detecting the conflicts at commit time, i.e., applying lazy conflict detection. They employed linear and exponential back-off functions to recover from transactional conflicts. Below, we first briefly review the design techniques used in the non-blocking STM, using the design outlined by Harris et al., followed by their blocking STM implementation, and then report the performance related comparison among these two STMs.

Non-blocking STM

Harris et al. [20] argue that helping proposed by Shavit et al. [7] is expensive due to the heavy contention it generates. Therefore, instead of helping mechanism, they proposed a mechanism based on locks stealing with merge (during stealing phase) and redo (during lock release phase) operations as we explain in the following.

At commit time, a transaction attempts to acquire all the locks it requires to get exclusive access to its memory locations. If a transaction has acquired all its locks, but not yet written the new values, other conflicting transaction can steal locks from it, using the new value that is to be written. However, if the transaction has not yet acquired all its locks, the stealer transaction may abort it before attempting to acquire its own locks.

In the stealing phase, the stealer transaction merges the transaction entries from the conflicting transaction, which we refer to it as victim, to its own. However, stale updates may occur during the lock release phase of a victim transaction, where the stealing transaction makes updates to the shared memory locations before the victim transaction does, and then the victim transaction makes its stale updates to those locations, overwriting the most recent updates made by the stealer transaction.

To avoid such stale updates, when the victim transaction discovers a potential risk of stale updates, on detecting a stolen lock, it redoes the updates to the memory locations corresponding to the stolen lock, using the new value for the location stored in stealer's transaction entries, only if the stealer is no longer active. Therefore, the most recent update is always performed to the shared memory locations.

Blocking STM

In the blocking STM, during the execution of a transaction, for each memory location accessed for read, a check is performed to see whether its lock has been taken by other transaction or not. If the lock is taken, the transaction waits until it the lock is released before transferring its value to its local memory. This check is done once again to make sure the lock is still free, and has not been taken yet, otherwise, the check is repeated until the lock is free again. For writes to the memory locations, however, since the STM uses lazy version management, all the memory updates are applied locally using the redo log corresponding to the thread performing transaction.

At commit time, the blocking STM transaction needs to ensure that no other thread block has changed any of the memory locations accessed earlier, that is, read and/or written, before it can write back the updates to shared memory. Therefore, each of the accessed memory locations is checked, and if the original value read matches the current value in shared memory, its lock is acquired using a compare-and-swap (CAS) atomic operation. Recall that the CAS operation will only succeed as long as the lock status has not changed in the meantime. Any failure in acquiring the locks, or the lock has already been taken, causes the transaction to abort. Once all locks have been successfully acquired, the updates are written back to the shared memory, followed by releasing the locks. Note that, in case of abort, all the already acquired locks should also be released for functional correctness as well as data consistency.

To evaluate and compare the effectiveness of STMs, Cederman et al. [8] carried out their experiments with 2 different contention levels. In the first version, the threads in test applications perform some local work before executing the transactions, leading to a low contention scenario, while in the second version, the high contention scenario is achieved, by having most of the work performed through transactions.

Overall, based on their reported result, at a low level of contention, the blocking and the non-blocking STMs both scale similarly, that is, when the number of thread-blocks increases, the number of operations per millisecond grows equally. In addition, the average number of aborts per transaction is about the same in both STMs. With higher contention, however, the performance with respect to the number of operations per millisecond is better using the non-blocking STM, and hence, the corresponding test applications display higher level of scalability.

Given different back-off schemes, there were no significant change in performance and abort rate. This could be attributed to the hardware scheduler’s behavior and the fact that there is only one transaction per thread-block. Consequently, regardless of limited concurrency level in the STMs proposed by Cederman et al., providing additional progress guarantees, such as obstruction-freeness, appears effective to improve performance, while a blocking STM is simpler to implement.

2.3.2 Hierarchical Validation STM

Xu et al. [9] proposed an STM system on GPU which consists of a *hierarchical validation* (HV) technique and an *encounter-time lock-sorting* mechanism with commit-time locking, which we refer to as H*STM*, attempting to address the major challenges of using fine-gained locking on GPU: (1) by ensuring good scalability with respect to the massive multithreading, and (2) by preventing livelocks caused by the SIMT execution paradigm of GPUs. Unlike block level STM we discussed earlier which suffers from limited concurrency, HSTM supports per-thread transactions for each thread within a thread block. Furthermore, the HSTM uses the early conflict detection for reads and late detection for writes, while implementing the lazy version management.

The hierarchical validation technique combines *timestamp-based validation* (TBV) with *value-based validation* (VBV), two commonly used conflict detection strategies. At the cost of a slight increase in memory requirements, and of the potential contention on a shared timestamp counter, performing the TBV significantly reduces the high overhead of using the VBV alone, whereas the VBV is helpful to eliminate false conflicts caused by timestamp-based validation.

The TBV is implemented by associating a timestamp with each group of shared memory locations (memory stripe) to indicate when these location was last modified. A transaction also acquires a timestamp indicating the time, referred to as linearization point, when its effects become visible to the rest of the system. Thus, the comparisons of the order of execution is made possible, providing the HSTM system an opportunity to skip validation when the memory location was last modified before the transaction's linearization point.

In GPU, the back-off mechanism is not ideal to avoid livelocks, because threads within the same warp cannot wait for different delays due to the lockstep execution. To adopt a back-off mechanism specific to GPU, when lock acquisition is required, first, the threads within a warp try to acquire locks in parallel, and then those which failed try to acquire locks again sequentially. Hence, livelocking within a warp is avoided by sequentially locking, which could cause a bottleneck during commit and thus degrade performance. Therefore, in the HSTM, the encounter-time lock-sorting together with commit-time locking is used so that the transactions can acquire the locks in the same order. Hence, livelock-freedom is ensured, and no backoff mechanism is required. Unfortunately, the HSTM suffers from poor performance and low level of scalability which can be attributed to high overhead of its validation mechanism.

2.3.3 Lightweight STM

The inadequate scalability which previous STM implementations exhibit is due to high overheads in their transaction management, especially in conflict detection. To address this problem, Holey et al. [10] proposed a low overhead, lightweight STM *(LSTM)* so that the applications could scale satisfactorily to thousands of threads available in GPU-based systems. Given a program, the LSTM exploits the characteristics of the program to either trade−off conflict detection accuracy with detection overheads or avoid tracking entry updates on reads by making reads invisible to other transactions. Holey et al. showed, with their proposed techniques, programs with varying levels of contention can scale to thousands of threads on GPU.

In order to implement LSTM, Holey et al. [10] suggested two flavors of transactional support exploiting an overall transaction's property calculated for a program: *visibility.* This property can be calculated by dividing the total number of writes followed by reads to the memory locations over the total number of reads. Note than this property can be variable during the runtime of a program. The LSTM uses eager version management along with eager conflict detection on writes or eager/lazy conflict detection on reads, depending on the transaction's visibility, performed during a transaction.

In eager conflict detection, a transaction can abort other transaction while it gets aborted later, therefore, the livelock is not avoidable. For instance, two transactions can keep on aborting each other, leading to a livelock. To avoid livelocks a backoff mechanism is utilized in the LSTM by adding a variable delay before restarting aborted transactions to ensure forward progress. In the following, we briefly discuss both abovementioned transactional supports.

The pessimistic transactional support of LSTM specifically targets the applications that in good portion of their execution time, the same memory locations are read and then written within the same transaction, that is, they own high values for visibility property we discussed earlier. Such support reduces the conflict detection overhead which is possible by tracking the speculative reads and writes equally, leading to a quite simple, yet low overhead conflict detection mechanism. I.e., in addition to regular read−write conflicts, a conflict can also be detected even if two threads read the same memory location, exploiting the fact that if a transaction in a program reads a location, it will most likely write to the same location. Therefore, the read conflicts are detected eagerly, leading to early release of the computation and memory resources.

Conversely, in programs which do not exhibit such memory access behavior, they will incur high overhead costs due to false conflicts. I.e., given pessimistic transactional support, the performance of LSTM could hurt when a program has higher number of reads than writes within its transactions. This is caused by fewer actual updates to the memory locations, therefore, majority of the conflicts detected are false, leading to low performance.

To implement pessimistic support, the tracking entry for each memory location records which thread has accessed it by using the id of thread wherein a conflict is detected by a single CAS atomic operation.

In second transactional support, which is known as invisible read, the conflicts on reads are optimistically detected/validated in lazy fashion at commit time. I.e.., speculative reads are invisible to other transactions, and because they do not modify the tracking entries, this leads to faster transactional reads. This support suits the programs which during their execution, only small portion of reads to memory locations are followed by writes to the same locations, yielding to higher performance. Recall that both transactional support designs detect conflicts on writes eagerly.

Each tracking entry used for detecting dependence violations in invisible read support is consist of version number and lock status, we refer to the combination as version locks. The version lock can be implemented using an unsigned integer to represent lock status with the lease significant bit and the rest of the bits for version number. Thus, using versioned locks speculative reads remain invisible to other transactions, while writes are still visible. Note that the version number is locally stored in transaction private memory and compared with the shared memory at commit time.

Holey et al. evaluated their proposed techniques against the fine-grained locking and two state-of-the-art STMs previously discussed, from the standpoint of overheads and scalability. They also used the benchmarks with varying degree of contention, number of memory operations, and transaction sizes for their experimentations. Their results show that the proposed STM support considerably outperforms the earlier STM works, while achieving performance comparable to fine grained locking. Specially, their techniques perform better than other STM designs which specifically target high−contention workloads, on programs with higher conflict rates.

Overall, when critical sections contribute to small portion of the total execution time, that is, the programs spend significant time executing the native code, both transactional supports show low overhead cost. I.e., if the time spent in transactions is relatively short, the STM overheads can be amortized by non−transactional execution in the program. Moreover, among pessimistic and invisible read transactional supports, pessimistic version performs better under high contention and when same data are read first and then written in the transaction.

Recall that in the STM design of Cederman et al. [8], a transaction executes at a thread block level rather than at a thread level, that is, the threads within a thread block do not fall into race conditions; this assumption, of course, is not practical in order to fully utilize the system multiprocessors as SIMT engines. To have a fair comparison, however, Holey et al. modified this design so that each thread within a thread block can take independent transaction(s). Subsequently, they showed LSTM design scales better as the number of GPU threads increases compared to both STM implementations of Cederman et al. and HSTM.

Lower scalability of both Cederman et al.'s STM and HSTM can be attributed to consistency checks performed on reads, which hinders their performance. In Cederman et al.'s STM the read operation waits until a concurrent write from other transaction to the same location is complete, while HSTM performs incremental validation after each read that involves timestamp checking followed by value−based validation for all reads performed earlier within a transaction. Note that even though these STMs employ lazy version management, but even under high contention the eager mechanisms of LSTM outperforms, which is due to lightweight conflict detection mechanisms.

3 TRANSACTIONAL-MEMORY-BASED IMPLEMENTATION OF MINIMUM SPANNING FOREST ALGORITHM ON GPU

Minimum Spanning Tree (MST) is used in many real world applications, e.g., distributed networks [21], VLSI design, and medical imaging [22]. Parallelizing of MST calculation on GPUs exemplifies the challenges of executing irregular applications on these platforms. An MST algorithm on an undirected connected graph finds a minimally weighted subset of the graph edges that connects all the vertices without any cycle. If the graph has several connected components, then it involves calculating the Minimum Spanning Forest (MSF) which constitutes one MST for each connected component.

Among the different proposed algorithms for MSF, the Borouvka's algorithm [3] has the most expressed parallelism. It calculates a subset of the graph edges for each connected subgraph of the forest that spans all the vertices with minimum cost. As illustrated in Algorithm 3.1, for each such subgraph G, all the vertices are initialized as individual components. The algorithm iterates through the components and connects each component to another component with a minimum cost path from the component. The iterative process continues until only one component is left.

```
Input: an undirected graph G
 // each component $c_i$ initially contains one vertex $v_i$
T = {$c_1$,$c_2$,$c_3$…$c_k$}
while (number of components in T) >1 {
   for each component $c_i$ in T{ // edge discovery phase
      S = {};
      for each Vertex $v_j$ in $c_i$ {
         e = minOutComponentEdge ($v_j$);
         S.add(e);
```

```
            }
            T.add (minEdge(S));
        } // end of edge discovery phase
        mergeComponents(T); // merge phase
    } // end of while loop
    Output: T is minimum spanning tree of Graph G
```

Algorithm 3.1. Borouvka's MST algorithm

Over the past few years, there have been several implementations of morph algorithms on GPUs, which are relatively complex and non-scalable. Vineet et al [4] implemented the Borouvka's MST algorithm on GPU, using CUDA primitives such as scan, segmented scan, and split on a recursive formulation of Borouvka's algorithm. However, their approach is not only complicated but also doesn't scale well to larger graphs due to the data restrictions, e.g., by imposing some restrictions on the edge weights and the number of vertices. Though it shows substantial performance gain for some types of graphs, their approach is not applicable to other larger graphs, e.g., for the majority of the randomly generated large graphs in section 3.2 of this chapter.

On the other hand, Nasre et al. [5] implemented the Borouvka's algorithm on GPU, avoiding the synchronization all together. However, they proposed rather complex algorithmic techniques, such as subgraph addition together with multiple phases deviating from the phases of the original algorithm, e.g., it included a phase to break the formation of cycles among components to prevent race conditions. Their approach, however, did not prove to be quite efficient as we show in our experimental result which could be due to the extra work required to avoid synchronization. In section 3.2, we do a further comparison with their work.

Kang et al [23] designed and implemented an STM based Borouvka's algorithm on a multicore CPU system with 64 cores. They concluded that satisfactory performance, compared to the fastest known sequential implementation, can be achieved only if further solutions are available to lower the overhead of transactional memory, e.g., by having a transactional memory at the hardware level. As we will see in the experimental results (section 3.2), their work has some additional limitations for randomly generated large graphs due to restrictions in the number of disconnected components.

In this chapter, we demonstrate an efficient transactional-memory-based design and implementation of the Borouvka’s algorithm on GPUs. In the first phase of the Borouvka’s algorithm (edge discovery), we extract certain algebraic properties of the computation (e.g. monotonicity) [6] to exploit lock-free synchronizations (i.e., synchronizations using atomic instructions). The edge discovery phase is monotonic due to ever decreasing values produced in finding the MSF edges. However, the second phase of the algorithm (merge phase) lacks such algebraic properties and hence we utilize STM as a synchronization mechanism in this phase.

In addition to STM, we apply different optimization techniques like “warp segmentation” [12] in our implementation to maximize GPU utilization and load balancing, which are found to significantly enhance efficiency and performance.

The rest of this chapter is organized as follows: Section 3.1 elaborates our transactional-memory-based design and implementation of Borouvka's algorithm on GPUs. The experimental evaluations are discussed in section 3.2. Finally, section 3.3 concludes the chapter with a summary of our work.

3.1 Transactional-Memory-based Minimum Spanning Forest

In this section we discuss the design and implementation of a transactional-memory-based MSF algorithm. We partition the Borouvka’s algorithm (Algorithm 3.1) into two main phases: edge discovery and merge. In the edge discovery phase, there exists an algebraic property of monotonicity which helps implement lock-free synchronizations using atomic operations. This is further elaborated in the following. In the merge phase, however, there are no such algebraic properties and hence we resort to an STM-based synchronization mechanism to overcome race conditions.

In the edge discovery phase, each vertex in a component initially finds its minimum-weight edge (edge suggestion) to other components. Based on the original Borouvka’s algorithm (Algorithm 3.1), the edge suggestions of different vertices in a component are stored in a set from which the least expensive edge is selected. In contrast, in our implementation, all the vertices of a component can write their edge suggestions to a shared location in a lock-free approach. We exploit the disjoint-set data structure to represent the components, wherein each component C contains a disjoint-subset of vertices; a tree data structure represents such a disjoint-subset with its root vertex as the representative of C. Additionally, the “union” operation merges two subsets into one by attaching the root of the tree associated with one subset into another, and the “find” operation locates the subset’s representative of a constituent vertex by traversing its corresponding tree. The representative maintains the shared location. A vertex’s edge suggestion is compared with its representative’s shared location and if such suggestion is less expensive, then the representative’s content is replaced.

Since we are only interested in retaining the least expensive of edge suggestions in each subsequent comparison, the edge weight in a component representative monotonically decreases. As such, the corresponding algebraic property of monotonicity is exploited to discover the least expensive edge suggestion for a component. If an atomic-free approach is used (i.e., without the use of atomic operations like “compare and swap”) to handle the critical race in writing to a shared location in a component representative, we are required to repeat the edge discovery phase until there is no further change in the content of any component representative. However, through our experiments we noticed that such atomic-free approach could result in more work due to increased number of iterations in finding the MSF of moderate size graphs. Hence we use atomic “compare and swap” operations in the edge discovery phase because the focus of our work is on large graphs.

In the merge phase, we use the minimum-weight edges, which are discovered and collected in each component representative, to connect their endpoint components (disjoint subsets) via disjoint-set “union” operation. Since components may run into race to join with each other, synchronizing the merge operations is inevitable. We resort to an STM-based synchronization mechanism by implementing a priority-based STM for the merge phase. The STM facilitates a natural mapping of the merge phase to the implementation code, e.g., variable sized transactions are handled seamlessly as opposed to fine-grained synchronization using locks.

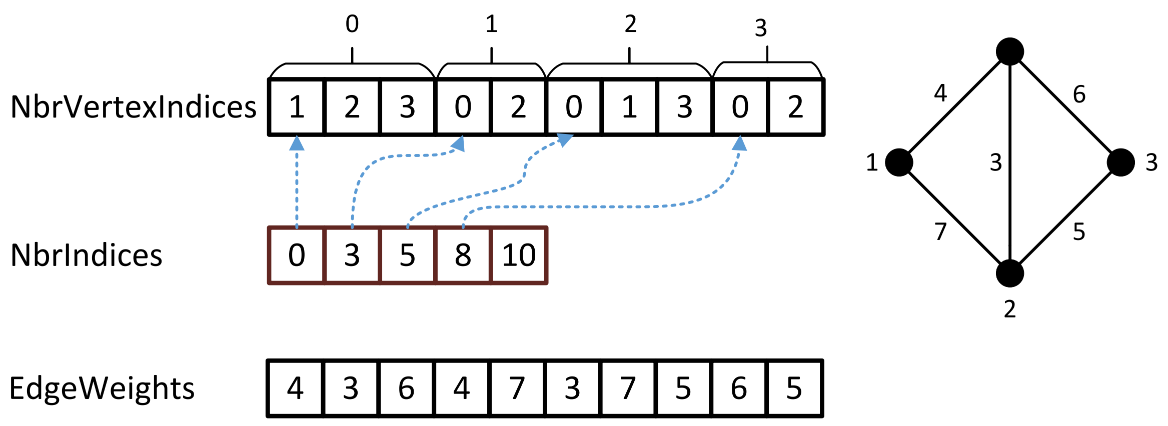


Figure 3.1. A graph with 4 vertices and 5 edges in the CSR format

Two optimization techniques are applied at the end of the merge phase: pruning and flattering. Pruning reduces the workload of the edge discovery phase in the subsequent iterations by eliminating the edges whose endpoints are located in the same component. In flattening, each vertex's parent is updated to point to the root vertex (i.e., the component representative), reducing the cost of disjoin-set "find" operations in future invocations.

To minimize the global memory overhead for the graph representation on the GPU, we use the Compressed Sparse Representation (CSR) format. Figure 3.1 shows the 3 arrays that constitute CSR:

- NbrVertexIndices: stores the adjacency list of graph's vertices.
- NbrIndices: contains the indices of the adjacency set of each vertex in NbrVertexIndices.
- EdgeWeights: holds the weights of the edges corresponding to the vertices in NbrVertexIndices.

In the following subsections, we discuss the edge discovery and merge phases in detail.

3.1.1 Edge Discovery Phase

The main task of the edge discovery phase is to determine the minimum-weight edge from each component of the graph connecting it to other components. In each component, it first finds the minimum-weight edge (edge suggestion) for each vertex connecting it to adjacent vertices in other components. This is followed by finding the minimum of these edge suggestions: discovered MST edge.

To implement this phase on a GPU, the naive approach is to have each thread in charge of processing one vertex and all its incident edges. However, in CUDA-capable GPUs, a set of 32 contiguous threads of a warp execute in the SIMD fashion. So, due to different number of neighbors of each vertex in the graph, the threads in the warp can have dissimilar workloads, resulting in intra-warp load imbalances and hence performance degradation.

To address the above problem, in our approach we utilize the "warp segmentation" technique [12] in which the adjacency lists of the vertices are grouped in different sized segments. Each warp is assigned to 32 consecutive vertices (we call them warp-assigned vertices) and their adjacency lists, which are copied to the fast shared memory in a warp to maximize the processing speed. However, since these adjacency lists could not fit into the shared memory of a warp, they are streamed into a shared buffer in chunks of 32 neighbors.

Hence, all the threads in the warp collaborate iteratively in processing the neighbors of the warp-assigned vertices, resulting in increased load balance. During each iteration, threads of a warp perform parallel reduction to find the minimum-weight edges incident on the warp-assigned vertices. Upon finding such minimum-weight edge in each vertex (i.e. edge suggestion), it is compared with the current value of the component's representative which monotonically retains the least expensive of such edge suggestions.

Figure 3.2 shows the result of edge discovery performed on a graph with three components: A, B and C. The figure also shows the MST edges discovered in each component by retaining the least expensive edge suggestions made from the component's vertices. For example, the least expensive edge is selected among all edge suggestions made to component A's representative (i.e. vertex 2), which is illustrated by pointers from vertices 1, 2, 3 and 4 to vertex 2. The final suggestion of component A is the MST edge incident on vertex 2 (in component A) connecting to vertex 5 (in component B).

Algorithm 3.2 illustrates the edge discovery phase. Initially, a shared memory buffer _Shared_Suggestions is allocated as a placeholder for all incident edges of each warp-assigned vertex (line 2). The second shared memory buffer is allocated to fetch indices of NbrIndices corresponding to the warp assigned vertices (lines 3-4). Each warp thread determines the region within NbrVertexIndices and EdgedValues arrays that belongs to the 32 assigned vertices (lines

5-6). Finally, the warp threads iterate in a for-loop over all the edges incident on the warp-assigned vertices to determine the minimum-weight edges (lines 7-18).

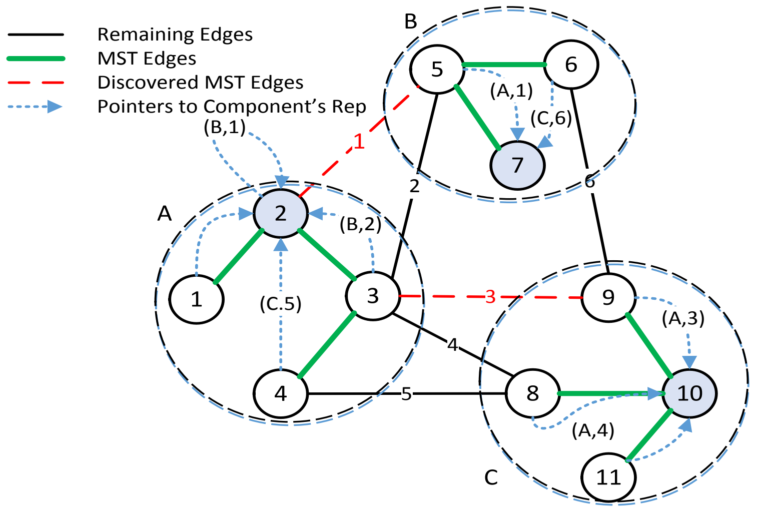


Figure 3.2. Result of edge discovery phase in example graph

```
1. parallel-for warp  {
2. declare _Shared_Suggestions[blockDim] of Edge;
3. declare _Shared_NbrIndices[blockDim] of Integer;
4. _Shared_NbrIndices [warpThreadBlockOffset +laneID]
                        = NbrIndices [GlobalThreadID];
5. startEdgeIndex // boundary start
            = _Shared_NbrIndices [warpThreadBlockOffset];
6. endEdgeIndex //boundary end
            = NbrIndices [warpGlobalOffset + 32];
7. for (edgeIndex = startEdgeIndex + laneID;
                        edgeIndex < endEdgeIndex; edgeIndex += 32) {
8.   segment = getWarpSegParams(edgeIndex, endEdgeIndex);
9.   if segment.warp_assigned_vertex is not active continue;
10.        reduceWithinWarpSeg (edgeIndex, segment);
11.        if  segment.inSegId == 0  // first thread in a segment
12.          if  segment.warp_assigned_vertex has no edge suggestion
13.            deactivate(segment.warp_assigned_vertex);
14.          else {
15.            rootVertex = find ( segment.belongingVertex );
16.             suggest(rootVertex, segment.belongingVertex);
17.            }
18.      } // for loop end
19. } // parallel-for warp end
```

Algorithm 3.2. MST edge discovery operation

All the consecutive edges belonging to a warp-assigned vertex form a segment within a warp, wherein the parallel reduction is performed to find the minimum-weight edge. Therefore, during the for-loop iterations, each warp thread first determines which of the 32 warp assigned vertices owns the current edge using a binary search within _Shared_NbrIndices; this is followed by detecting the segment size and its index (inSegId) inside the segment (line 8). Having the segment information, the minimum-weight edge is determined by parallel reduction (line 10).

Following the reduction step, the first thread of each segment compares the minimum-weight edge, stored in _Shared_Suggestion buffer, with the rootVertex (i.e. representative of the component) of its warp-assigned vertex (lines 15-16). However before such comparison, if the current warp-assigned vertex does not have any edge suggestion left, it will be deactivated (line 13). Eliminating such vertices from the components improves the algorithm efficiency by reducing useless computations in the following edge discovery phases (line 9). In performing the comparison (via suggest function), the suggested weight is compared to the current weight in the root vertex and the minimum of both is retained in global memory through an atomic operation (e.g. "compare and swap"). Subsequent comparisons with each component's vertices monotonically decreases the current suggested weight in each component until the minimum of such weights is found.

3.1.2 Merge Phase

In the merge phase, components of the graph are merged together to form larger components using the MST edges discovered in previous phase (edge discovery). For this purpose, each component's representative (root vertex) is queried to get its discovered MST edge, from which the neighbor vertex of the other component can be located. Subsequently, the root vertex of the other component is located using the neighbor vertex. Finally two components are merged into one by selecting one of the root vertices as the root vertex of newly constructed component.

During the merge phase, components may run into critical race to join with each other. To avoid such race conditions, we exploit synchronized transaction to perform merge within the critical section; we call it merge transaction. The merge transaction atomically reads all the vertices involved to locate the root vertex of the other component, followed by updating them to point to the root vertex of the new component. Here, we design and implement a lightweight priority-based STM as a synchronization mechanism for the merge transactions.

Figure 3.3 shows the merge phase following the edge discovery phase for the example graph in Figure 3.2. First, in the race among components A, B and C to merge with each other (A to merge with B, B to merge with A, and C to merge with A), component B successfully proceeds resulting in component new B which is constructed encompassing all the vertices from the previous components A and B. As the result of the merge, vertex 2, previously the root vertex in component A, now points to vertex 7 which is the component new B's root vertex (Figure 3.3(a)).

Later, component C proceeds to merge with the newly created component, new B, which now contains the vertex 3, previously a member of component A. The vertex 3 is the neighbor vertex located using the discovered MST edge for component C. The thread processing the root vertex 10 of component C starts a new transaction which first locates the root vertex (vertex 7) of vertex 3 and then updates vertex 7, previously the root vertex of component B, to point to component C's root vertex (vertex 10). Finally, within the same transaction, vertices 3 and 2 are updated to point to vertex 10, which is the component new C's root vertex.

To support STM based transactions, we include an unsigned integer as a tracking entry in each vertex of the graph. The most significant bit of the tracking entry indicates the lock status and the remaining bits are used for the priority of the thread performing a transaction. We define threadID value as the priority for a thread where the highest value indicates the lowest priority. Each thread allocates a local array of vertices to keep track of all the vertices participating in an STM transaction. The size of the local array is a constant L, i.e., the longest transaction size. A transaction of size greater than L aborts and restarts until its size is lowered as the side effect of merging of other components.

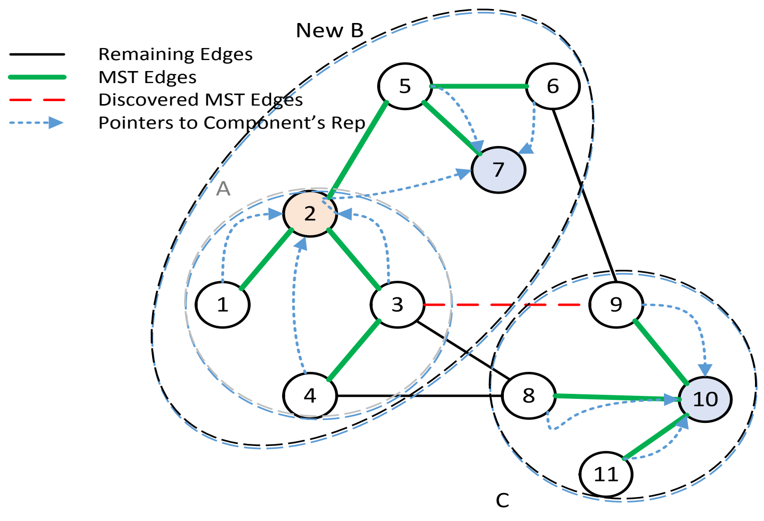


Figure 3.3(a). Merging component B with A to create new B

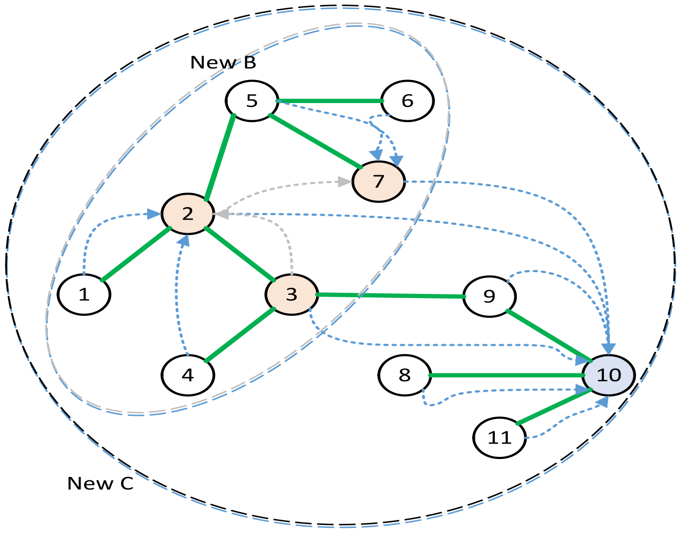


Figure 3.3(b). Merging component C with new B

Figure 3.3. Merge in example graph following the edge discovery phase

If a thread inside a transaction requires to access a vertex, it first attempts to atomically mark its tracking entry by updating its priority value to its own id. Marking is successful when the tracking entry is not locked (lock bit is 0) and its priority is lower than thread's; otherwise the transaction aborts and restarts. Following a successful marking, any modification to the vertex is kept in the local array until the end of transaction. At the end of marking and modification phase, a thread rechecks the tracking entry of each vertex it requires and if its priority hasn't changed, it attempts to acquire its lock. The transaction aborts if any of its required vertices' priorities has been changed or if any of the lock bits have been set by other transactions with higher priority. Once all the locks are acquired successfully, the local memory changes are applied to the global memory via the commit function and the locks are released. Before releasing the locks, we need to invoke the __threadfence instruction [13] to ensure that the changes reach to the global memory, thus avoiding any inconsistency caused by the weak memory consistency model [24] in CUDA GPUs.

```
1.  function mark(){
2.  globalLockPriority = vertices [index].lockPriority;
3.  if globalLockPriority > thread.priority
4.      ||  isLocked(globalLockPriority) {
5.    rollback();
6.    return false;
7.  }
8.  if  thread.priority > globalLockPriority
9.      && !isLocked(globalLockPriority)  {
10.   oldValue = atomicCAS(&vertices[index].lockPriority,
11.       globalLockPriority, thread.priority);
12.   if gLockPriority != oldValue
13.     rollback();
14.   else
15.     return true;
16. }
17. return false; }
```

Algorithm 3.3. Mark function used in priority based STM

Algorithm 3.3 illustrates the mark function used in our priority-based STM. At the beginning of the mark function, a thread's transaction tries to mark its required vertex. If the vertex is already locked (lock bit is 1) or the thread's priority is lower than the current priority, the transaction rollbacks (lines 2-6); otherwise the thread attempts to replace its priority using the “compare and swap” (CAS) [13] atomic operation (lines 8-11). If the CAS operation is not successful then the transaction rollbacks via rollback function (lines 12-13).

If a transaction fails within the marking phase then the tracking entries are reset by invoking the rollback function. The rollback function replaces the priority values of the tracking entries with the lowest priority through the CAS atomic operation, which implicitly releases the locks by resetting the lock bits. Resetting of tracking entries gives the subsequent transactions chances to proceed with marking in a non-blocking fashion.

In our implementation, the priority part of the tracking entry is set to a maximum integer value (indicating the lowest priority) using the CAS atomic operation to signify that there is no previous marking of the vertex. This is in contrast to the work of Shen et al. [11] where a separate bit is dedicated to represent the marking status for a memory location. Moreover, we improved the performance of our STM implementation by removing the overhead of versioning in [11], which is required to enable the invisible reads; this is possible because in our algorithm each read of a memory location (vertex) is followed by a write operation.

Algorithm 3.4 illustrates the merge phase in more detail. Initially each vertex is assigned to a thread. Before a thread starts a transaction, it first checks whether the root vertex is still a valid root vertex, i.e., its component has not yet been merged with other components; so it stops further processing if the current vertex is not a valid root vertex (lines 2-4). Subsequently, by performing transactional disjoint-set find function, it can terminate the transaction if the root vertex has already been attempted to be merged by other transactions (lines 6-12). Hence, by performing the above checks, we avoid the overhead of transactions as far as possible.

The transactional disjoint-set union function is invoked using the root vertex, which holds information about the neighbor vertex in the other component as an end-vertex of the discovered MST edge. The union function internally invokes the find function to locate the other component’s root vertex before updating it to point to the root vertex of the newly merged component (line 16). Finally, if all the locks of the marked vertices are acquired successfully through the acquireLocks function (line 18), the local changes are applied to the global memory via the commit function and the transaction completes successfully. Similar to the rollback function, tracking entries are reset within the commit function following updates to the global memory.

```
parallel-for warp {
parentIndex = vertices [vertexID].parentIndex;
if parentIndex != vertexID
  return;
while true { // beginning of transaction
  (rootVertex, boolean)= find (vertexID);
  if  boolean { // marking was successful
    rootVertexIndex = rootVertex.vertexIndex;
    if  rootVertexIndex != vertexID {
    /* component  has merged before */
      rollback();
      break; // transaction completed
    }
  } else
    continue; // to restart transaction when marking failed
  boolean = union(rootVertex);
  if  boolean  // marking was successful in union
    boolean = acquireLocks();
  if  !boolean // locks acquisition failed
    continue; // restart transaction
  else
    commit();
  break; // transaction completed
  } // end while
  } // parallel-for warp end
```

Algorithm 3.4. Merge operation using STM transactions

3.1.3 Optimization

In each iteration of the MSF algorithm, the graph is pruned and flattened at the end of the merge phase so that the workload of the both edge discovery and merge phases decreases in the following iterations. The rationale behind pruning is that, in an undirected graph, each edge is repeated two times in the adjacency list with opposite directions. Pruning eliminates all the edges whose opposite direction edges have already been added to the MSF. Furthermore, it removes all the edges whose end vertices are already placed in the same component after a merging phase.

Pruning is realized using a kernel dedicated to remove such edges, which exploits "warp segmentation" [12] for load balancing of the edge-processing threads. During pruning, we set the value of a removed edge to *max* (upper bound on edge weight) so that this edge will be disregarded in the edge discovery phase of the following iterations.

Flattening performs the "path compression" of the disjoint-set data structure. It utilizes the "pointer doubling" technique, replacing the parent of each vertex by its grandparent. To have a better load balance of the running threads, at each iteration of flattening, only one level of "pointer doubling" is applied to each vertex. Note that this operation is not requiring any synchronization as we exploit the idempotency algebraic property.

Both pruning and flattening operations are executed in parallel as there is no dependency between them. The two operations are terminated when there are no more changes in the graph, i.e., all required edges are removed and all vertices are pointing to their root vertices.

3.2 Experimental Evaluations

In this section, we evaluate the performance of our transactional-memory-based MSF implemented using CUDA on two different GPUs, and compare it with both the serial and the STM-based implementation on a multicore CPU. At the end, we also compare our experimental results with those of Nasre et al. as reported in [5].

Experimental Setup

To evaluate our transactional-memory-based MSF, we choose some of the sparse graphs of large and moderate sizes from USA road networks [25] as summarized in Table 3.1 In addition to these real world graphs, we generate 4 other large sparse graphs using the random GTgraph

Graph Generator [26]; the sizes of two of the graphs are chosen to be close to the real world western USA road network and for the other two, the number of edges is halved. Moreover, we generate multiple denser random/R-MAT [26] graphs which are used to demonstrate the speedup and scalability of our work.

We carried out the experiments using two GPUs of different strengths: a server/workstation based NVIDIA Tesla K40 with 2880 stream cores, 288GB/sec of memory bandwidth, 12GB of GDDR4 memory, and core boost clock of 875 MHz; and a desktop based Quadro K1200 with 512 steam cores, 80GB/sec of memory bandwidth, 4GB of GDDR5 memory, and core boost clock of 1124 MHz. For comparison purposes, we used Boost graph library's [27] implementation for fastest sequential algorithm on our Intel Core i7-6700 CPU with 8 cores and 16GB of DDR4 memory. Moreover, on the same CPU we ran the multicore STM-based implementation [23]. For all compilations, we used gcc 4.4.7 with O2 optimization level on the Scientific Linux 6.8 operation system.

Table 3.1. USA Road Networks, Random, and R-MAT graphs

Graph Name	Description	N.Vertices	N.Edges
USA	Full USA Road Network	23.9M	58M
W	Western USA Road Network	6.2M	15M
E	Western USA Road Network	3.5M	8.7M
FLA	Florida Road Network	1M	2.7M
NY	New York City Road Network	264K	733K
RND1	random graph	6.2M	15M
R-MAT1	Power-law graph	6.2M	15M
RND2	random graph	6.2M	7.6M
R-MAT2	Power-law graph	6.2M	7.6M

Experimental Results

Table 3.2 summarizes the execution time comparisons of our implementation on Tesla GPU (CUDA) versus sequential and multi-core STM-based [23] implementations. Table 3.2 also includes abort rates of transactions in our GPU-based implementation.

Referring to the size of each graph, we can infer that the larger graphs with lower abort rates achieved better speedups for sparse real world graphs, e.g., among the road networks from Table

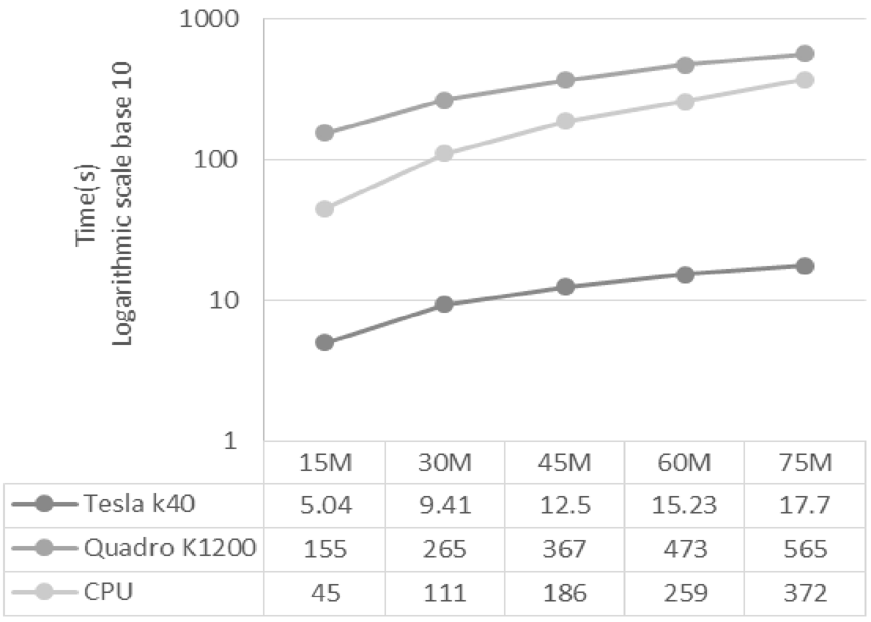
3.1, the graph named USA is the largest one which resulted in 4.52 times speedup relative to the serial implementation while showing relatively lower abort rate. For the same graph we achieved 2.69 times speedup relative to STM-based multi-core implementation. In contrast, the NY graph is the smallest sized among the road network graphs. However, it resulted in the highest abort rate (77%) and a lower speedup as compared to other larger real world graphs. In summary, for majority real world graphs, our implementation outperforms the serial and multi-core counterparts.

Table 3.2. K40 GPU versus serial and STM-based parallel on CPU

Graph Name	i7-6700 CPU		TeslaK40 GPU			
	Parallel(ms)	Serial(ms)	CUDA App(ms)	Abort Rate	Speedup vs Parallel	Speedup vs Serial
USA	36064	60546	13407	0.38	2.69	4.52
W	7980	14568	3451	0.38	2.31	4.22
E	4157	6581	1959	0.37	2.12	3.36
FLA	1082	1346	736	0.72	1.47	1.83
NY	301	278	217	0.77	1.39	1.28
RND1	*	47220	8133	0.01	*	5.81
R-MAT1	*	46678	6786	0.03	*	6.88
RND2	*	19360	5538	0.01	*	3.5
R-MAT2	*	19141	4791	0.03	*	4

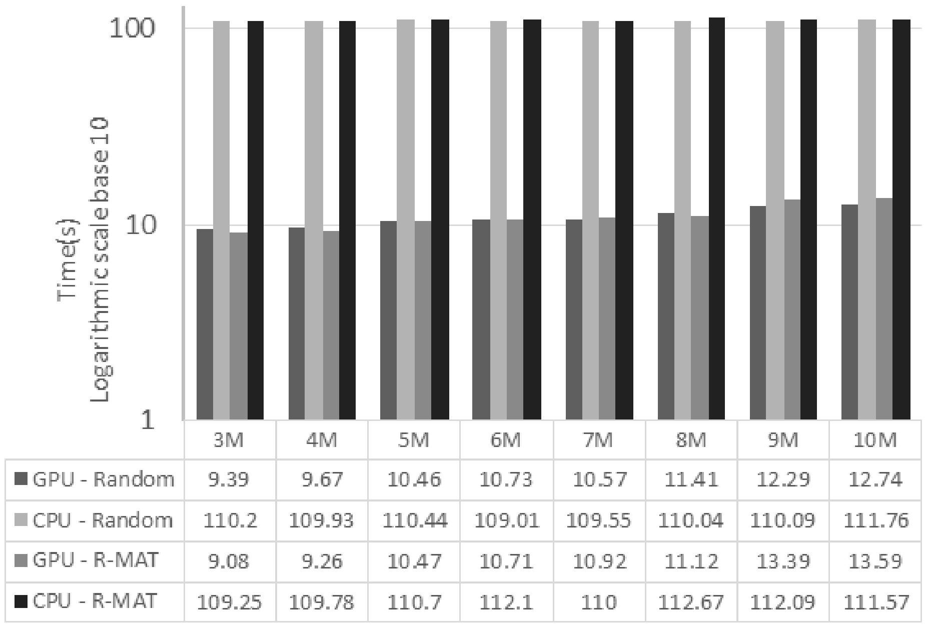
*. Failed to produce results

In random and R-MAT generated graphs, when the number of edges doubles, the better speedup is achieved, e.g. R-MAT1 with double the number of edges (15M) as compared to R-MAT2 (7.6M) achieved better speedup (6.88 versus 4). While both the graphs show negligible amounts of conflicts among the threads (shown by abort rates), the better performance is due to the higher degree of concurrency due to 2880 CUDA cores in the Tesla GPU. The STM-based implementation on CPU failed to produce any results, as the large number of MSTs due to many disconnected components in the generated graphs is a limiting factor.



	15M	30M	45M	60M	75M
Tesla k40	5.04	9.41	12.5	15.23	17.7
Quadro K1200	155	265	367	473	565
CPU	45	111	186	259	372

Figure 3.4. Comparison of execution time of CPU versus different GPUs



	3M	4M	5M	6M	7M	8M	9M	10M
GPU - Random	9.39	9.67	10.46	10.73	10.57	11.41	12.29	12.74
CPU - Random	110.2	109.93	110.44	109.01	109.55	110.04	110.09	111.76
GPU - R-MAT	9.08	9.26	10.47	10.71	10.92	11.12	13.39	13.59
CPU - R-MAT	109.25	109.78	110.7	112.1	110	112.67	112.09	111.57

Figure 3.5. Execution times on Tesla GPU versus serial on CPU for random/R-MAT graphs

In the next experiment, we use denser random graphs with 3M vertices and the number of edges ranging from 15M to 75M. As illustrated in Figure 3.4, the computational time on the

Tesla K40 GPU is considerably better than both desktop Quadro GPU and serial run on CPU. In a graph with 15M vertices and 75M edges, we achieved a speedup of 21 on Tesla GPU as compared to the fastest sequential algorithm. The better result on Tesla GPU as compared to that of Quadra GPU can be attributed to the scalability of the approach with larger GPUs with increasing number of cores (e.g. 2880 CUDA cores on a Tesla GPU as compared to 512 on a Quadro) in spite of increase in the amount of synchronization required due to increased degree of concurrency.

Subsequently, we generate random and R-MAT graphs of different number of vertices ranging from 3M to 10M and with the same number of edges (30M). Figure 3.5 illustrates the execution times on CPU and GPU. Increasing the number of vertices, while keeping the number of edges fixed, increases the sparseness of the graph; however it raises the execution time mainly due to the inevitable memory latency and extra work required in the merge phases with additional vertices. However, the performance of the transactional-memory-based GPU implementation is still significantly better than the serial implementation on CPU.

We were not able to evaluate the implementation of Nasre et al [5] for our input graphs because the code was not publicly available at the time of our experiments. Hence, in the following, we compare with the reported results. The experiments in [5] were performed on a Tesla C2070 GPU with 448 CUDA cores and 144GB/sec of memory bandwidth. In comparison, the number of CUDA cores and the bandwidth in K40 are respectively over 6 and 2 times greater than C2070. The results in [5] show a processing speed of 1.61M edges/sec for the largest roadmap graph in the data set, i.e., USA. In our experiments, the corresponding processing speed is 4.30M edges/sec when run on a Tesla K40 GPU. For the largest, densest graph in [5], i.e., a randomly generated R-MAT graph with 1M vertices and 8.3M edges (RMAT20), the reported processing speed is 0.31M edges/sec. In comparison, for the densest of our input graphs, i.e., an R-MAT graph with 3M vertices and 30M edges, we achieved a processing speed of 3.30M edges/sec. Interestingly, the number of vertices (edges) in our test graph is 3 (3.61) times greater, while the rate of processing of the edges is over 10 times higher. Finally, we performed a test on a generated graph with the same attributes as RMAT20 in [5], which resulted in processing of 3.6M edges/sec, i.e., over 11.6 times higher than that reported in [5].

3.3 Summary

In this chapter, we designed and implemented a transactional-memory-based Borouvka's MSF algorithm on GPU. MSF is a morph algorithm with irregular memory access pattern. We showed that we could reduce the gap between the original algorithm and its implementation through a transactional-memory-based approach. Extracting the algebraic properties of the algorithm in the first phase (edge discovery) helps implement fast lock free transactions. However, in the second phase (merge phase), due to lack of such algebraic properties, we implemented a priority based STM to synchronize its transactions which ensures the transactions to be both deadlock- and livelock-free. We also exploited several optimization techniques (e.g., pruning, flattening, and "warp segmentation") to improve performance.

4 TRANSACTIONAL-MEMORY-BASED IMPLEMENTATION OF GRAPH PARTITIONING ON GPU

Graph partitioning arises in different application areas such as VLSI design, task scheduling and scientific computation. The goal for a graph partitioner in such applications is to partition a graph into roughly equal sized parts while the edge connections between different parts are minimized. For example, to efficiently perform some of the parallel computations, a graph partitioner initially partitions the task interaction graph of the computation, in which the tasks and communications among them are modeled via vertices and edges, respectively. As a result, the tasks are fairly distributed among the processing units, leading to workload balance, and the data transferred via partition's cross-part edges are reduced. The graph partitioning is an NP-complete problem which makes it an attractive problem for researchers who are targeting for algorithms generating higher quality partitions with least runtime.

In past three decades, different approaches for graph partitioning have been proposed. Among these methods, spectral scheme derives the partition from the spectrum of the adjacency matrix. However, in the downside, spectral method comprises of an expensive, time consuming computation of eigenvector, corresponding to the second smallest eigenvalue. Geometric methods, on the other hand, are only applicable to graphs wherein the geometric information is available. Recently, however, a class of algorithms known as multilevel paradigm has gained popularity which aims for decreasing the partitioning time at the cost of somewhat decreased partition quality. For the rest of our discussions, we explore this class of graph partitioning algorithms.

The aim of this chapter is to show the implementation aspects for an efficient, transactional-memory-based graph partitioner on massively multithreaded GPUs. In doing so, the multilevel scheme is exploited as the main design strategy with the introduction of GPU specific operations, aiming to improve performance as compared to other parallel implementations. In short, multilevel scheme consists of three phases: coarsening, initial partitioning and un-coarsening. During the coarsening phase a large graph is successively reduced in size. This is achieved by matching the densely connected vertices followed by collapsing the matched vertices, producing an intermediate coarser graph which is used as an input for the next level coarsening phase. The coarsening terminates when the size of the coarsest graph is small enough so that the initial partitioning can be performed using a more expensive algorithm to obtain a partition of high quality. Then, in un-coarsening phase the resultant partition is projected to the immediate finer graph, followed by refinements made to improve its quality. This process is repeated until the final partition is achieved with the refinements applied to the original graph in last pass of the un-coarsening phase.

To efficiently parallelize the coarsening and un-coarsening phases of graph partitioning on GPUs, exploiting associativity algebraic property, we utilized the segmented scan primitives along with different high-level primitives derived from them, e.g. “sort” and “pack”. A segmented scan primitive is the general form of the scan primitive where the parallel scan operations are performed on arbitrary segments of the input array versus the entire array. The segments are commonly specified by a head-flag array where each segment is marked at its first element. These primitives are useful during contraction and pre-refinement stages performed at coarsening and un-coarsening, respectively. STM transactions are used during matching and refinement wherein synchronized access to shared data is required. Additionally, we utilize “warp segmentation” to maximize the throughput by reducing the load imbalance during marking processes. These are further elaborated in the following.

In the following, section 4.1 outlines the basic idea of graph partitioning. In section 4.2, a multi-threaded implementation on multicore [28] is discussed. The main structure of the algorithm used in our implementation uses this work as the baseline. In section 4.3, we review the GPU implementation of segmented sort and pack primitives. Finally we discuss the transactional-memory-based implementation of graph partitioning on GPUs in section 4.4.

4.1 Multilevel Graph Partitioning

In this section, we outline the basic idea of multilevel graph partitioning algorithm used as the basis of the design and implementation for partitioning of large graphs in parallel platforms. We start by giving the formal definition of k-way graph partitioning and then we go through the formulation of each phase of multilevel method.

Given a graph $G = (V, E)$ with $|V| = n$, wherein V and E are the set of all vertices and edges, respectively, the goal is to partition G into k disjoint parts: $V_1, V_2, V_3 \ldots V_k,$ such that each part contains roughly n/k vertices, and the number of edges whose endpoints resides in different parts is minimized. The set of such parts constitutes a partition for the input graph.

To account for graphs in which the weights are associated with vertices and edges, the k-way graph partitioning problem is extended to find a partition wherein the sum of vertex weights in each disjoint part V_i is roughly the same as others whereas the sum of edge weights for edges connecting different parts, which from now on we refer to as edgecut, is minimized. Besides, the set of all edges where each has endpoints residing on different parts of partition is called the *cut* in a graph partition. Moreover, the endpoints of the edges in cut are referred to as *boundary vertices*. The result of portioning is commonly represented by an array P of length n holding a value between 1 and k specifying the part number each vertex belongs to.

A common, simple solution to k-way partition problem is the recursive bisection. This method first performs the initial bisection (2-way partition) of V into 2 parts, and then recursively keep bisecting each part until the k-way partition is produced. Internally, the multilevel method is used to perform each bisection within the recursive algorithm.

The 3-phase structure of the graph bisection using multilevel approach is as the following. In coarsening phase, the graph G_0 is coarsened down to a graph G_m with substantially reduced number of vertices, that is, a sequence of smaller graphs, $G_1, G_2, G_3 \ldots G_m$, are generated such that $|V_0| > |V_1| > |V_2| > \ldots > |V_m|$. During initial partitioning phase, a 2-way partition of the coarse graph $G_m = (V_m, E_m)$ is computed such that each part contains roughly half of the vertices of original graph G_0. In un-coarsening phase, the resulting partition P_m is consecutively projected back to each finer graph $G_{m-1}, G_{m-2}, G_{m-3} \ldots G_2, G_1$ and G_0, whereas the refinement is performed at each step to reduce the edgecut, i.e., the partition P_m of G_m is projected back to G_0 by going through intermediate partitions, $P_{m-1}, P_{m-2} \ldots P_1, P_0$, with refinement performed at each transition. Figure 4.1 depicts the multilevel graph bisection. In the rest of this section we discuss each of these phases in more details.

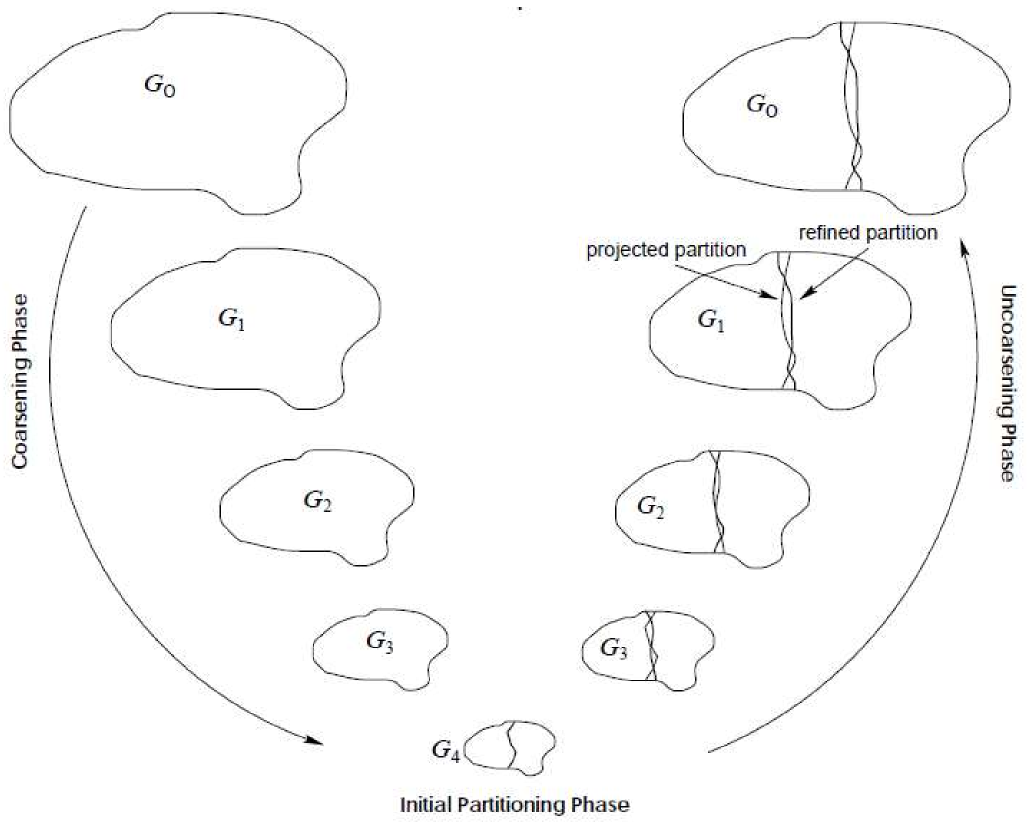


Figure 4.1. Multilevel graph bisection adopted from [32]

4.1.1 Coarsening

Coarsening of a graph is realized by the matching stage followed by the contraction. At each step of matching, each set of vertices V^v of graph G_i are matched followed by a collapse during contraction to form a single, coarse vertex v of the next denser graph G_{i+1}. Since the bisection of the coarsest graph G_m will be ultimately projected back to original graph G_0, it is essential to have a good estimation of the original graph G_0 in G_m. In order to achieve such an estimation, during contraction stage, the set of vertices V^v are merged into a coarse vertex v with its weight reflecting the weights of the vertices in V^v and its incident edges contains the edges connected to the vertices in V^v. This will ensure the weight and connectivity information is preserved in coarser graphs: G_1, G_2, G_3 ... G_m, which is required later in initial partitioning of G_m and each pass of un-coarsening to successively improve the quality of the partition.

In order to generate a coarser graph G_{i+1} from G_i, the adjacent vertices are collapsed to form coarse vertices in G_{i+1} based on the result of computation in the matching. In doing so, a coarse vertex is generated which its weight is set to the sum of the vertex weights in V^v, and then its incident edges are constructed by combining the edges connected to vertices in V^v as the following. At first, the edge connecting the vertices in V^v are removed. Then if some of the vertices in V^v share the same adjacent vertex u not in V^v, all incident edges on u connecting u to vertices in V^v are replaced by a new edge. Finally, the weight of the newly created edge is set to the sum of weights of the above incident edges.

A *matching* M_i of graph G_i is a set of edges with distinct matched endpoints. Given M_i, the next level coarser graph G_{i+1} can be induced by collapsing the matched vertices of G_i into coarse vertices, whereas the unmatched vertices remain intact in the coarser graph. Since the goal is to coarsen a graph into a graph as small as possible, the minimum number of unmatched vertices are desired; this leads to having M_i with greater size, i.e., the size of matching M_i has a direct effect on the size of the successive coarser graph, which indirectly reduces the number of coarsening passes. Hence, in the matching stage we aim for finding a matching with larger size: *maximal matching*. A matching is maximal if for any of the edges not present in the matching, at least one of its endpoints is matched, i.e., it is included in M_i.

A simple, yet efficient, approach to compute a maximal matching is *random matching* (*RM*) [29, 30]. This approach generates a maximal matching in a greedy fashion as the following. The vertices are randomly visited, and if a vertex v has not yet been matched, then one of its unmatched adjacent vertices, vertex u, is randomly selected, then its incident edge is included in the matching, and u is marked as matched. However, if no such a vertex u exists, vertex v remains unmatched in the final matching. The complexity of RM algorithm is $O(|E|)$. Since the high quality of a partition is achieved when the edgecut is minimized, RM cannot be an ideal choice.

It's easy to show that the total edge weight of the coarser graph G_{i+1} is reduced by the overall edge weight of the matching M_i. Consequently, a maximal matching M_i with large total edge weight is more beneficial to substantially reduce the edge weight of the coarser graph G_{i+1}. As discussed in [31], the coarser graph G_{i+1} with lower overall edge weight has smaller edgecut. For this reason, a maximal matching which contains edges with large weights is desired. *Heavy Edge Matching* (*HEM*) [32] is a technique to obtain such maximal matching. Similar to RM, HEM is a randomized algorithm wherein the vertices are visited randomly, but rather than randomly matching a vertex *v* with one of its unmatched adjacent vertices *U*, the vertex *v* is matched with a vertex *u* which is connected with by a heaviest edge among other edges connecting v to the vertices in *U*. The complexity of the HEM algorithm is $O(|E|)$, which is asymptotically similar to that of RM.

4.1.2 Initial Partitioning

In initial partitioning phase of the multilevel graph partitioning, the coarse graph G_m is bisected into two roughly equal size parts, generating partition P_m. This partition is used to partition the original graph G_0 because the weight associated with each vertex (edge) in V_m (E_m) reflects some distinct set of vertices (edges) in G_0. I.e., since the connectivity information together with the vertices weights of G_0 has been preserved during the coarsening phase, a good partition of much smaller coarse graph G_m applied to G_0, by going through intermediate graphs, generates a reasonably accurate partition in G_0.

Karypis et al. in [32] implemented an initial partitioning algorithm based on KL algorithm [33] with some modifications as we will be discussing below. KL Algorithm is an iterative algorithm to find the locally optimal 2-way partition (bisection) of the given graph G_m. However, the behavior of KL algorithm is affected by the quality of the initial bisection. Among different techniques to find a good initial bisection, *greedy graph growing partitioning (GGGP)* suggested in [32] is based on a strict breadth first search (BFS) which operates as the following: starting from a random vertex v, GGGP algorithm grows the area around v by inserting the vertices into the growing region of BFS if such vertices reduce the edgecut, giving the priorities to vertices with larger decrease. We may require to repeat performing the GGGP algorithm a few times, from different random vertices, before a suitable initial cut is discovered.

Given a decent initial bisection, in each pass of KL algorithm a subset of vertices from each part is found which when swapped will result in a partition with reduced edgecut. This process is repeated as long as such a subset can be found, otherwise the state of the computation has been reached to the local minimum, at which point no further improvement is possible. The KL algorithm is a local optimization algorithm, with restrictions to get out of local minima. Conversely, in a class of more expensive algorithms, known as hill-climbing, the search can continue to find the global minimum state by moving a group of vertices across a partition boundary at each point in time, therefore, a higher quality partition can be revealed.

Karypis et al. [32] followed the work of Fiduccia and Mattheyses [34] who reduced the runtime of KL algorithm by using appropriate data structures, reducing the complexity, from $O(|E|\ log_2(|E|))$ per iteration in KL algorithm, to $O(|E|)$. Additionally, Fiduccia and Mattheyses proposed a slight change in original KL algorithm by suggesting to move only one vertex across partition boundary at a time, versus swapping a pair of vertices of KL algorithm.

Given a 2-way partition P, the gain g_v of a vertex v is defined as the amount of edgecut reduction/increase resulted by moving v from one part of the partition to the other, that is, the difference between the sum of the edge weights for the incident edges on v belonging to the partition's cut, and the sum of the edge weights for the v's incident edges whose endpoints reside on the same part as v. Given this definition, if g_v is positive (negative), then the edgecut is reduced (increased) by g_v if v is moved to the opposite part of the partition. The gain g_v can be formulated as the following:

$$g_V = O_v - I_v$$
$$O_v = \Sigma\ weight(v,u)\ \forall (u,v) \in E : P[u]!=P[v]$$
$$I_v = \Sigma\ weight(v,u)\ \forall (u,v) \in E : P[u]=P[v]$$

Having the gain computed for each vertex, in each iteration the vertex with the largest gain from the larger part is selected, and moved to the other part. After the vertex move, the partition connectivity of each vertex u adjacent to v changes, hence, the associated gain is updated. Additionally, the moved vertex is marked so that it will not be considered for move in the same iteration.

While the original KL algorithm continues until all the vertices have been speculatively moved, the implementation in [32] terminates as soon as L number vertex moves with no edgecut decrease has reached. This considerably reduces the runtime of the process. Note that last L speculative moves are reverted if they in fact increase the edgecut rather than decreasing it.

4.1.3 Un-coarsening

During the un-coarsening phase, the partition P_m is successively projected back to each finer graph $G_{m-1}, G_{m-2}, G_{m-3} \ldots G_2, G_1,$ and G_0 , wherein the partition label of each coarse vertex v in G_{i+1} is assigned to the set of vertices V^v of next level finer graph G_i. Having the vertices in G_i with more degrees of freedom resulted from the projection step, the refinement step performs the swapping of partition's parts of such vertices to further reduce the edgecut in P_i.

There have been different refinement algorithms, which are essentially variants of the KL algorithm, proposed with good results in final partition quality of the graph partitioning. One of such refinement schemes, known as *boundary KL refinement* (*BKL*) introduced by Karypis et al. [32], performs more efficiently because it inserts only a subset of vertices based on their gain, as opposed to all vertices, in an appropriate, refinement-related data structure. The algorithm in [32] is based on the observation that only small number of vertices result in edgecut reduction, and such reduction mostly occurs in the first pass of the refinement. Moreover due to nature of refinement, most of the vertices are swapped along the boundary of a cut. Therefore, only boundary vertices are required to be inserted into the refinement-related data structure.

Furthermore, as the number of vertices is significantly smaller in coarser graphs, a refinement stage can take more passes to further reduce the edgecut with no significant performance degradation as opposed to finer graphs with higher number of vertices. Hence, an adaptive mechanism proposed in [32] during refinement is beneficial which adjusts the number of refinement passes based on the number of vertices in the intermediate graphs.

Recall that Karypis et al. also proposed to terminate a pass when there is no more decrease in edgecut after certain number of attempts (L). Even though reducing the number of the swapped vertices also helps reduce the runtime, the major improvement in asymptotic complexity is due to reduced number of vertices inserted into the data structure; hence, leading to the major impact in the overall runtime.

4.2 Multi-threaded Graph Partitioning

In this section we discuss the multi-threaded implementation (from now on we refer to it as mt-metis) [28] of multilevel graph partitioning using multicore CPUs based on shared memory architecture. The good experimental results of mt-metis are mainly achieved through applying barrier based synchronization in both matching and refinement stages, avoiding any fine grained synchronization overhead. In the following, we review the implementation of each major phase in mt-metis graph partitioning: coarsening, initial partitioning and un-coarsening, along with the data structures and algorithms used.

4.2.1 Coarsening

As discussed in section 4.1, to build the next-level coarser graph, each step of coarsening of the multilevel graph partitioning consists of vertex matching followed by graph contraction through collapsing the matched vertices into coarse vertices and merging the adjacency lists of such vertices.

Mt-metis parallelize the matching stage by first distributing the vertices of the graph among the threads; hence, each thread subsequently carries out the matching only for the local vertices which it owns. The HEM matching algorithm is also adopted by mt-metis to find a match for each vertex. This, however, requires the threads to read/write to the vertices assigned to other threads where the matched vertex is not a local vertex to a thread, leading to race conditions. In the following we discuss how mt-metis resolves such conflicts.

Initially, exploiting the shared memory architecture, a shared matching array *Mtchs* is created in which each element corresponds to a vertex of the graph. Hence, each thread is able to freely read any location in *Mtchs* in order to lookup the matching status of a vertex followed by writing to the same location read earlier if the matching has been determined.

Given the graph $G_i = (V_i, E_i)$, to remove the possibility of race conditions during the reads/writes from/to the locations in *Mtchs*, one approach is to employ the fine-grained locking as synchronization method. However, the excessive locking of fine-grained locking synchronization leads to poor performance. This approach can be slightly improved by synchronizing the write operations, i.e., excluding the read operations, and incurring the extra cost to ensure consistency.

To address the high overhead of the fine-grained locking mechanisms, mt-metis uses an approach which relies on the heuristic nature of matching as the following. As each thread matches two vertices, it may (if the match is not local) populate memory locations in *Mtchs* corresponding to both local and non-local vertices. However, in absence of locking, the matching for some of these vertices could be lost by the writes made by other threads. As a result, upon completion of the matching performed by all threads, a second step to resolve the write conflicts is required. We call this correction step.

During correction step, each thread goes through the list of its local vertices and for any vertex v with an asymmetrical matching *(Mtchs[v]* = *u* and *Mtchs[u]* = *v*), it matches it with itself. Since the number of vertices in the graph is considerably greater than the number of threads, the asymmetrical incidences are insignificant; therefore, asymmetrical incidences will not impact the size of the maximal matching M_i. Figure 4.2 depicts an example demonstrating this 2-step matching process performed on a graph with 8 vertices.

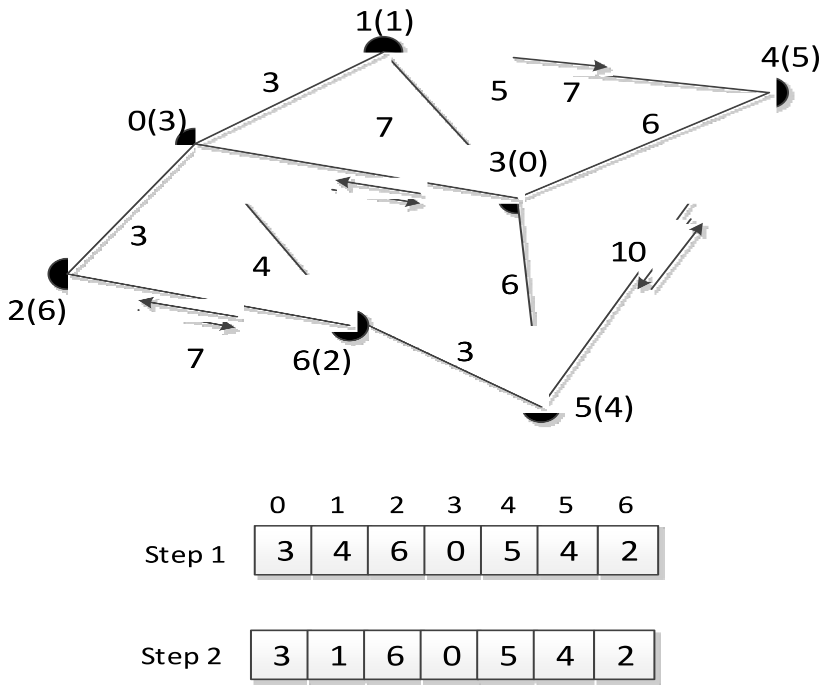


Figure 4.2. Matching stage of coarsening phase

Once the final matching is produced, a 4-step parallel computation [41] is performed over *Mtchs* in order to assign coarse vertex numbers for each pair of matched vertices, generating vertex mapping array *CMaps* in which each element corresponding to a vertex points to the coarse vertex in next level graph. This 4-step parallel computation which is performed prior to contraction step is as the following. Starting with *CMaps* whose elements are initialized to 0, for each vertex v with index value *i*, the *i*th element in *CMaps* is set to *1* if the value of *i* is less than or equal to the index value of its matched vertex *u*, that is:

i <= Match[i]

A parallel prefix-sum is then performed over *CMaps* followed by a process to subtract one from each element. Finally, for each *v* if the value of its index *i* is greater than the index value of its matched vertex *u*, *i*th element in *CMaps* is set to the value of the element with index value of *u*, that is:

CMaps[i] = CMaps[Match[i]] : i > Mtchs[i]

The following example demonstrating this 4-step process performed on the result of matching from Figure 4.2:

```
 0 1 2 3 4 5 6       # Vertices
[3 1 6 0 5 4 2]    # Step 1:    Input matching array
[1 1 1 0 1 0 0]    # Step 2:    Set each CMaps's element to 1 if i <= Match[i]
[1 2 3 3 4 4 4]    # Step 3.1: Inclusive prefix sum on CMaps
[0 1 2 2 3 3 3]    # Step 3.2: Subtract 1 from each of the CMaps's elements
[0 1 2 0 3 3 2]    # Step 4:    Set CMaps[i] to CMaps[Match[i]] if  i > Match[i]
```

Having both *Mtchs* and *CMaps* array properly populated with above procedures, the final operation (contraction) is to produce the coarser graph G_{i+1}. To do that, having the coarse vertices in G_{i+1} divided among the threads, each thread merges the adjacent lists of the set of matched vertices V^v in G_i for each coarse vertex *v* in G_{i+1} following the same procedure outlined in section 4.1.1.

4.2.2 Initial Partitioning

Although the problem size of initial partitioning is small, the parallelization of it using mt-metis is still effective to reduce its total runtime compared to that of serial version. Two methods of parallelization were experimented in mt-metis are *parallel bisecting* and *parallel k-sectioning* which we explain below.

Parallel bisecting is a recursive algorithm wherein each phase consists of a bi-section of the input graph until k-way partition is constructed, i.e., given *p* threads, at the beginning, each thread bisects the input coarse graph G_m, generating G_L and G_R. This is followed by a synchronization among threads to perform a min-reduce so that the 2-way partition with the minimum edgecut can be selected for further processing. Afterwards, the threads are split into two groups, each receiving either G_L or G_R as input to bisect their part, followed by selecting the best 2-way partition for the next level. This recursive process ends after $log_2(k)$ levels of a binary tree, producing the k parts of the partition. In parallel k-sectioning, each thread individually produces the k-way partition via recursive bisection, and then all threads synchronize so that the best partition among all of the threads is selected via a min-reduce operation.

As shown in [28], the parallel k-sectioning is faster, in spite of having less number of parallel tasks, compared to parallel bisecting. This is attributed to the cost of the barrier synchronizations required in parallel bisecting.

4.2.3 Un-coarsening

Similar to the original multilevel algorithm discussed in section 4.1, mt-metis performs a projection followed by a refinement at each step of un-coarsening phase. However, unlike the original algorithm in which the initial partition is a 2-way partition, mt-metis starts with a k-way initial partition to encourage the concurrency among threads, i.e., the threads work on different parts of the partition at the same time leading to proper utilization of multiple cores. This change, on the other hand, other than complicating the gain calculation, requires to broadcast updates to the neighbors when the corresponding vertices move during refinement. To avoid these side effects, one can employ an estimated value for gain to simplify its calculation. As discussed in [35], the external cost value can be computed by dividing the value edgecut against all external partition's parts by the root square of total number of external partition's parts connected to a boundary vertex.

The parallelization of projection stage is straightforward as the threads can work in isolations and with no data sharing among them. However, the refinement stage is serial in nature, thus to make a concurrent refinement, some relaxation to the original algorithm and data structures is necessary. On the downside, parallelizing the refinement process causes race conditions because the concurrent threads may access the same parts of the input graph at the same time: vertices, edges and the partition's parts. Recall that in the optimal refinement algorithm we discussed in section 4.1.3, in each iteration the vertex with the largest gain from larger part of the bisection is selected, and moved to the other part. Therefore, given a k-way partition at each pass of refinement, care should be taken to assure the weight balance of the partition is maintained.

To avoid data consistency issues of refinement-related data, one way of synchronization is through fine-grained locking. This is done by locking the partition's parts involved in vertex moves to ensure no partition's part exceeds the allowable size leading to an imbalance. However, this approach leads to poor performance mainly due to the bottleneck caused by the small number of partition's parts. Moreover, the excessive locking also degrades the performance when threads work on disjoint parts of a partition. To address the problems of fine-gained locking, mt-metis introduced heuristics based on barrier synchronization. In the following, first we discuss the relaxation made to the original algorithm to be able to perform a concurrent refinement, and then we outline the heuristics incorporated to reduce the cost of synchronization.

One way to implement the refinement algorithm is to use a priority queue as an appropriate data structure to store the boundary vertices based on their precomputed gains. As we discussed in section 4.1, we are only concerned with boundary vertices as opposed to all the vertices of the graph. One way of implementing the appropriate data structure for refinement-related data is priority queue, however, rather than constructing a single global priority queue one can create local, per-thread priority queues for the boundary vertices. The benefit of such relaxation technique is to make a parallel refinement stage to reduce its overall runtime, i.e., the threads working on disjoint queues in parallel lead to increased parallelism and speed up. Note that even though the local priority queues are not as effective as the global one, the overall impact on the partition quality is minimal, as shown in [28].

One issue that may arise using this implementation is that if two boundary vertices connected by a quite heavy edge swap partition's parts, depending on the weights other incident edges, this can lead to an increase in the edgecut. In mt-metis this issue is addressed by limiting the vertices to move across partition boundaries in only one direction at a time, i.e., at each pass of the refinement, the vertices are considered to move alternately from a partition's part with lower label number to higher one and vice versa.

Once a thread de-queue a vertex v from its priority queue, it finds the best eligible partition's part V_i as the destination of the move. V_i is eligible if its weight plus the weight of v is under the maximum tolerable amount, and the best partition's part is the one with the largest weight of the per-adjacent partition's part cut, that is, the sum of the weights of edges connecting v to the V_i is the maximum.

To avoid race conditions, mt-metis assigns an update buffer to each thread to receive messages for pending/committed vertex moves initiated by other threads. Moreover, as we explain below, the barriers are used as the means of synchronization for the operations of threads. Each thread also uses a local "set" data structure to communicate the potential changes in the partition's parts weights with other threads. Each element of this set represents a partition's part to reflect the total associated weight following such wright increases/decreases in partition's parts after vertex moves.

After a thread removes a vertex v from its local priority queue, it determines to which part of partition it should move v to. Assuming the source and destination part are denoted by V_S and V_D, respectively, this move can have a potential weight increase w in V_D, and the decrease of the same amount in V_S. Since the vertex moves impact the partition connectivity of its adjacent vertices, the thread initiating such move will require to update the gains of these adjacent vertices. If the adjacent vertices are local, the pending changes are stored locally, whereas the pending changes to the non-local adjacent vertices are stored as messages in the buffers of the threads owning these vertices.

After fixed number of pending vertex moves completed, the threads synchronize at a barrier, so they can communicate the potential weight changes to partition's parts with each other; this is to ensure that the balance of partition's parts are preserved. Upon completion of the communication, at next barrier point, the threads can decide whether to commit the change in the partition's connectivity of their local vertices as the result of moves, or to revert the moves violating the partition balance. The threads again synchronize at next barrier, so they can update their local vertices' partition connectivity for the moves committed by other threads. This process is repeated until all of the priority queues are emptied. Refinement is terminated after a certain number of passes, or when no vertex is moved in the last refinement pass.

4.3 Segmented Sort and Pack

Segmented scan is the building block we use to efficiently implement the transactional-memory-based graph partitioning algorithm. Thus, in this section, we first discuss the algorithms required to implement the segmented scan primitive, starting from the basic (unsegmented) *scan* primitive for having a similar main structure, and then, we outline high-level primitives: "enumerate"*, "copy"* and *"split-and-segment"* built on top of the segmented scan primitive. Finally we discuss how these high-level primitives are used to implement the "sort" and "pack" operations applied to segmented input of adjacency lists arrays used in the graph partitioning implementation (section 4.4).

4.3.1 Segmented Scan

The scan primitive requires global knowledge of all data elements as input array and an associative binary function $\oplus$ with an *identity* element to operate on these elements. Associative binary operations include addition [with identity *0*], *min*, *max*, logical *AND*, and logical *OR*. If the input is an array of elements such as:

$[a_0, a_1, a_2, a_3 \ldots]$

then an *exclusive* scan primitive outputs an array of:

$[i, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2 \ldots]$.

An *inclusive* scan, in contrast, produces the output of:

$[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, a_0 \oplus a_1 \oplus a_2 \oplus a_3 \ldots]$

wherein an inclusive scan is implemented by an extra step: adding the input array to the exclusive output array.

```
1. for d = 0 to $\log_2 n-1$ do
2.     for all k = 0 to n−1 by $2^{d+1}$ in parallel do
3.         x[k+$2^{d+1}$−1]=x[k+$2^d$ −1] + x[k+$2^{d+1}$−1]
```

Algorithm 4.1. The up-sweep (reduce) phase of a work-efficient sum scan algorithm

```
1. x[n−1]=0
2. for d = $\log_2 n-1$ down to 0 do
3.     for all k = 0 to n−1 by $2^{d+1}$ in parallel do
4.         t = x[k+$2^d$−1]
5.         x[k+$2^d$−1]=x[k+$2^{d+1}$−1]
```

```
6.        $x[k+2^{d+1}-1]=t + x[k+2^{d+1}-1]$
```

Algorithm 4.2. The down-sweep phase of a work-efficient parallel sum scan algorithm

In the following, we briefly discuss the details of the work-efficient scan formulation of Sengupta et al. [36] implemented on GPU using CUDA. Unsegmented scan of n elements requires two passes over the array: "*reduce*" and "*down-sweep*", as shown in Algorithm 4.1 and 4.2, respectively, wherein each thread processes two elements at a time. The "reduce" navigates a binary tree, with $log_2(n)$ levels, from its leaves to the root in order to compute partial-sums for internal nodes. While having the partial-sums, the "down-sweep" computes the final result at leaves navigating the tree from the root down to the leaves. Each "reduce" or "down-sweep" operation requires $log_2(n)$ parallel steps. The overall work complexity is $O(n)$ because the amount of work is reduced in half at each step. The behavior of the "reduce" and "down-sweep" is illustrated for an input array of 8 elements in Figure 4.3 and 4.4. Notice that in Figure 4.4 the first step zeros the last element of the array.

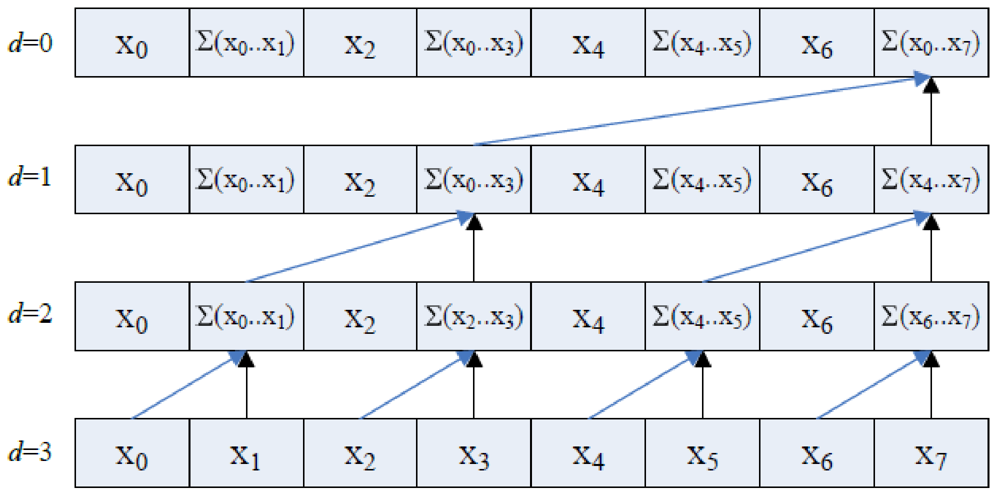


Figure 4.3. Up-sweep phase of work-efficient sum scan algorithm adopted from [36]

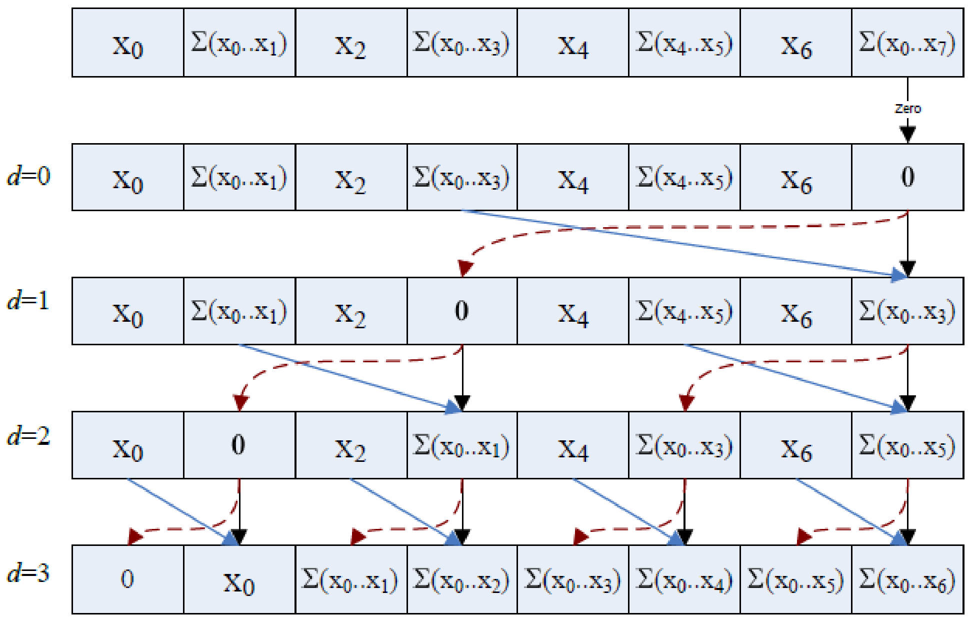


Figure 4.4. Down-sweep phase of work-efficient sum scan algorithm adopted from [36]

As the number of elements can easily exceed the maximum number allowable for a single thread block, the partial-sums of the input array is initially computed by each thread block. These results are used as inputs to a second-level recursive scan wherein each element of its output is then uniformly added to all elements of the corresponding block in the first-level scan. To be able to process the input arrays of large size, this process recursively continues until there is only one thread block left.

The segmented scan primitive [37] in GPU is a generalized form of the scan primitive which performs parallel scans on arbitrary sized *segments* of the input array as opposed to entire input array. The segments are commonly identified by a head flag array with the same size as the input array in which each segment is marked by its first element in a corresponding element in head flag array. The work-efficient segmented scan implementation on GPU [38] has the same complexity as scan: $O(n)$, but it's three times slower.

Segmented scan formulation is the extension of work-efficient scan [36], wherein Sengupta et al. slightly altered the "reduce" and "down-sweep" phases of the scan algorithm so that it could be efficiently implemented on GPU. In the following we briefly examine the segmented scan algorithm using "sum", also known as prefix-sum, as the binary scan operation; however any other binary associative operator can seamlessly be applied to this algorithm.

```
1. for $d$ = 1 to $\log_2 n-1$ do
2.     for all $k$ = 0 to $n-1$ by $2^{d+1}$ in parallel do
3.         if $f[k+2^{d+1}-1]$ is not set then
4.             $x[k+2^{d+1}-1]=x[k+2^{d}-1] + x[k+2^{d+1}-1]$
5.         $f[k+2^{d+1}-1]= f[k+2^{d}-1] \mid f[k+2^{d+1}-1]$
```

Algorithm 4.3. Reduce (up-sweep) phase of a segmented scan algorithm

Algorithm 4.3 presents the "reduce" phase of the segmented scan. Similar to "reduce" in scan, the tree is traversed from the leaves to the root of a binary tree. However, in addition to the partial-sums, it computes the partial *OR* flags by applying the logical *OR* of two input head flags while traversing the tree (line 5). Notice that in Algorithm 4.3 variable *x* denotes the partial-sums and *f* denotes the partial *OR* flags.

The "down-sweep" phase is shown in Algorithm 4.4. To compute the final segmented scan output, the "down-sweep" phase, similar to the scan, navigates the tree from the root to leaves using the partial-sums induced from the "reduce" phase. Similar to scan, in the segmented scan, the right child is computed as the sum of the value of its parent *p* and the former value of *p*'s left sibling (lines 10-11) except for the following conditions: If the flag in the right position of *p* is set in the head flag array then right child is set to *0*, the identity element, (lines 6-7) and if the partial *OR* stored in *p*'s position is set, the right child is set to the former value of *p*'s left sibling (lines 8-9).

In multi-block segmented scan, a second-level segmented scan is performed before the "down-sweep" starts. The last elements of partial sum and partial *OR* tree of each block are used as inputs to the second-level segmented scan. In addition, the first element of the head flag array of each block is received as the third input. Upon completion of the second-level segmented scan, the "down-sweep" receives the corresponding element of second-level segmented scan's output as the last element of each block. I.e., the last element of K^{th} block is set to the K^{th} element in output of the second-level segmented scan. In contrast, in the scan, the last element of each block is set to *0*, and the partial-sums are propagated among blocks by a uniform add.

```
1. $x[n-1]=0$
2. for $d$ = $\log_2 n-1$ down to 0 do
3.     for all $k$ = 0 to $n-1$ by $2^{d+1}$ in parallel do
```

```
4.        t=x[k+2^d-1]
5.        x[k+2^d-1]=x[k+2^{d+1}-1]
6.        if fi[k+2^d] is set then
7.           x[k+2^{d+1}-1]=0
8.        else if f [k+2^d -1] is set then
9.           x[k+2^{d+1}-1]=t
10. else
11.    x[k+2^{d+1}-1]=t + x[k+2^{d+1}-1]
12. Unset flag f [k+2^d -1]
```

Algorithm 4.4. Down-sweep phase of a segmented scan algorithm

4.3.2 Primitives Built On the top of Scan

In the following, we outline some of the high-level primitives [38] built on top of the segmented scan: "*enumerate*", "*copy*" and "*split-and-segment*". These high-level primitives are the building blocks of "pack" and "sort" operations utilized in the GPU implementation of the transactional-memory-based graph partitioning algorithm.

Enumerate

The "enumerate" operates on an input array of data elements along with a head flag array which contains a true/false "boolean" value for each element. The output of an "enumerate" primitive is an array with the same size as input wherein each output element corresponding to an input element contains the total number of elements to the left of that element in input array with their "boolean" values set to true. The extension of "enumerate" to a segmented input arrays operates on segments rather than the entire array, for example:

enumerate ([t f f t f t t] [t f t f]) = [0 1 1 1 2 2 3] [0 1 1 2]

To compute "enumerate", exclusive segmented scan is performed on a temporary array each element set to *0* (false) or *1* (true) corresponding to the "boolean" value of the element in the head flag array. The "enumerate" is the first step in "pack" (compact) operation. The "pack" is useful when one is interested in preserving the elements with a common attribute in an input array, e.g., if some elements are marked as true for an input array, upon completion of "enumerate" step, each true element holds the address of a location in the output array. At the end of “pack” operation, all elements marked as true are stored in the corresponding locations to the addresses they hold.

Copy

The “copy” operation in its generalized form copies the element at the head (or tail) of the segment to all other elements in that segment, for example:

copy ([c b a] [d e]) = [c c c] [d d]

The “copy” is implemented by setting all non-head elements to 0 in a temporary array and performing a segmented inclusive scan on that array.

Split-and-segment

The “*split”* primitive takes an input array of elements to partition it into two parts, i.e., bisect it, with all the false elements on one part of the output and all the true elements on the other part. However, the “split-and-segment” operating on a segmented input array performs an independent split on each segment. In the following example the letters: “t” and “f” next to data elements represent the “boolean” values of true and false, respectively:

split-and-segment ([at bf cf] [df et ff]) = [bf cf] [at] [df ff] [et]

To derive the addresses of true elements, Sengupta et al. [38] used additional computation, so that the computation requires only one pass of scan (“enumerate”). The following example depicts their formulation using an input array of 7 elements:

[t f t f f t f] # Initial input array
[0 1 0 1 1 0 1] # *E* = set *1* for false elements
[0 0 1 1 2 3 3] # *F* = “enumerate” with false=*1*
4 # *NF* = add the last elements in *E* and *F* arrays
# *NF* now holds the total no. of false values
[0 1 2 3 4 5 6] # Thread *IDs*
[4 5 5 6 6 6 7] # *T = ID - F + NF*
[4 0 5 1 2 6 3] # Address = if *E* is *1* then set the value of *F* else *T*

Segmented quicksort [38] (we refer to it as “sort”) is another useful application of segmented scan. The “sort” operation in a segmented input array is used to order the elements of each segment in parallel. The “sort” is an iterative algorithm formulated as the following. A selected pivot element in each segment, e.g., this could be the first element in the segment, is distributed across a segment via “copy” primitive, and then comparing each element with the pivot, a segmented head flag array is generated. Note that on alternating passes the comparison is performed by either greater than or greater-than-or-equal. Having the head flag array, the subsequent “split-and-segment” operation bisects each segment of the input array, moving the smaller elements to the head of the array and the larger elements to the tail. The algorithm is repeated until there is only one element in each segment. The following example shows the quicksort performed with two passes wherein each pass requires two segmented scans: one for the pivot “copy” and the other for “split-and-segment” operation:

```
[5 6 7 4 3]     # Initial input array
[5 5 5 5 5]     # Distribute pivot across segment via “copy”
[f t t f f]     # Set true if Input > pivot else false
[5 3 4][7 6]  # Split-and-segment
[5 5 5][7 7]  # Distribute pivot across segment via “copy”
[t f f][t f]    # Set true if Input >= pivot else false
[3 4 5][6 7]  # Split-and-segment
```

4.4 Transactional-Memory-based graph partitioning

In this section, we discuss the implementation of the transactional-memory-based graph partitioning on GPU. As discussed before, multilevel scheme consists of three phases: coarsening, initial partitioning and un-coarsening. In the following we briefly explain the techniques used to implement the coarsening and un-coarsening phases on GPU.

In the coarsening phase, the original graph G_0 is successively reduced in size producing the sequence of intermediate coarser graphs, with each pass comprising of matching stage followed by contraction stage. In the matching stage, first using “warp segmentation” for each vertex a matching candidate is found applying the HEM method, and then using the STM transactions the result of the matching candidates are stored in the matched vertices. In the event of conflicts between transactions, underlying STM system will help resolve such conflicts by allowing only one of the transactions to proceed and roll back the others. This is followed by generating the mapping array which for each vertex contains the corresponding vertex in the coarser graph.

During the contraction stage, the next level coarser graph is generated using the result of the matching from the preceding matching stage. In doing so, the adjacency list of all the vertices are first stored in a temporary array such that the matched vertices will have their adjacency lists reside next to each other, creating new segments of such adjacency lists. Subsequently, performing the "sort" operation based on the coarse vertex, the vertices with common the coarse vertices reside next to each other within these new segments. Finally, using the "warp segmentation" technique and "pack" operation we only keep distinct vertices, generating the adjacency list of the coarser graph. The coarsening phase continues until the size of a coarse graph is small enough so that we can execute the initial partitioning using a more expensive computation such as mt-metis on a multicore platform to obtain an initial partition of high quality.

Starting with an initial partition, in each pass of the un-coarsening phase the partition of current graph is projected back to the immediate finer graph, within projection stage, followed by refinement stage to improve the quality of partition in the finer graph. The projection stage is carried out by mapping the partition's part label (number) assigned to each coarse vertex v: $P_i[v]$ to the set of matched vertices V^v in next level finer graphs G_{i-1}. This stage can be performed in parallel, with no synchronization required, wherein each GPU thread operates on a vertex assigning its part number using its corresponding coarse vertex's part number. Each refinement stage includes multiple iterations to compute/update the gain for boundary vertices followed by moving the vertices with highest gains to other parts of partition. Since the priority queue is not a appropriate data structure to be used in GPU, being serial in nature, to properly utilize the hardware threads we mimic such queue's behavior. To do that, we go through iterations with each iteration only covering a restricted range of gain values such that initially, given a range with high gain values, the vertices with the highest gains are moved and in the following iterations, the vertices with lower gains are moved by extending the range to lower gain values.

To initially compute or update the gain for each boundary vertex v, the vertices of its corresponding adjacency list are grouped into segments based on the common partition's parts these vertices are residing in so that we can compute per-part edgecut. In this computation we utilize the segmented scan based operations ("sort" and prefix-sum) and the "warp segmentation" technique.

Moving a boundary vertex *v* from one part of the partition to other part involves invalidating the affected adjacent boundary vertices, updating the total weight of both source and destination parts, inactivating *v* to avoid further move in current refinement iteration and updating *v*' partition's part number to the new part. Note that these operations should all be performed together to maintain the data consistency, requiring us to utilize STM transaction to apply all the required changes.

In the following, section 4.4.1 outlines the data structure used to represent graphs in our implementation. Implementations of the coarsening and un-coarsening phases are detailed in sections 4.4.2 and 4.4.3 respectively. Furthermore, we perform the cost analysis in each of these sections.

4.4.1 Graph Data Structure

The data structure used to represent graph $G = (V, E)$ consists of three arrays following the CSR format: The first array, called *Vrtcs*, stores information related to the vertices and their adjacency list addresses, the second array, we refer to as *NbrVrtxIndices*, stores the adjacency lists of the vertices; each adjacency list of a vertex in *NbrVrtxIndices* commonly represents a segment on which the segmented scan primitives operate. The third array, we call *EdgeWeights*, holds the weights of the edges corresponding to the vertices in *NbrVrtxIndices*. For each vertex $v \in V$, *Vrtcs[v]* contains the following quantities:

- *p*: index to the vertex which *v* is being matched with
- *w*: the weight of *v*
- *noOfEdges*: the size of the adjacency list, i.e., the degree of *v*
- *indOfEdges*: the index into *NbrVrtxIndices* that is starting position of adjacency list corresponding to *v*

4.4.2 Coarsening

Given a graph $G_i = (V_i, E_i)$, each pass of the coarsening phase consists of matching stage followed by contraction. During matching, first for each vertex *u* a match is found, and then each set of matched vertices V^v in G_i is mapped to a coarse vertex *v* in the next level coarser graph

G_{i+1}. The graph G_{i+1} will be generated in contraction stage by merging the matched vertices and their associated adjacency list, constructing coarser vertices and edges.

Recall that as in mt-metis implemented for a multicore platform, the lower number of threads, as compared to the number of vertices, makes it feasible to apply the heuristic matching. Using this matching each thread freely matches its local vertices, and then, joins to other threads at a barrier point. This, however, leads to minimal asymmetric matches which threads resolve in isolation after the barrier point. Nevertheless, such heuristic is not applicable to the GPUs due to significantly higher number of threads in these SIMD platforms as opposed to multicore; therefore, we resort to STM transactions as a less costly alternative. To find and store the match condidates for each vertex, similar to mt-metis, we opt HEM technique combined with "warp segmentation" for performance. In doing so, an edge with maximum weight is found in each adjacency list using parallel reduction. Subsequently, having the match candidates, the match for each vertex is finalized using the STM transactions.

In each pass of contraction stage, the vertices and edges of G_{i+1} are constructed by merging each set of matched vertices in V^v to a coarse vertex v of G_{i+1} along with combining the edges incident on the vertices in V^v into the coarse edges incident on v. In contraction stage, exploiting the associativity as algebraic property, the segmented scan based operations such as prefix-sum, "sort" and "pack" are utilize to achieve maximum concurrency and efficiency. Moreover, the "warp segmentation" is utilized to mark the head flag array before the "pack" operation. In the following each of these stages: matching and contraction, is elaborated in more details.

In matching, for each vertex, using HEM technique, a candidate match is found. The match in HEM is an unmatched adjacent vertex connected with via the heaviest incident edge, if such adjacent vertex exists. However, we relax the requirement of being unmatched vertex as the actual matching is performed later, as we discuss below, via STM transactions. In doing so, initially the "warp segmentation" with an internal parallel reduction with *max* as associative operation is used so as to find the match for each vertex v applying HEM method. This is followed by writing the matched candidate in the matched vertices. However, the write conflicts may occur when storing the result of the match for two vertices marked as matched either with each other or with a third vertex at the same time. This requires us to use some sort of synchronization to address the race conditions, i.e., to resolve the conflicts.

One way to perform synchronization is via the fine-grained locking; however, this method is quite costly and leads to poor performance in GPU. The high overhead of fine-grained locking is mainly due the high number of GPU threads competing to acquire locks, e.g., using spin locks, which unnecessarily wastes the memory bandwidth. Therefore, we utilize STM as a less costly alternative. We opt the lightweight priority based STM implementation to manage the conflicting transactions. Successful execution of an STM transaction links the matched vertices using the match vertex field *p* described above, i.e., *p* will contain the index to its matched vertex, in the each of the matched vertices, or its own index if no match found.

Similar to mt-metis, to perform the matching and contraction operations, the *Mtchs* and *CMaps* arrays are utilized. Recall that the *Mtchs* stores the result of matching carried out by STM transactions and the *CMaps* stores the result of mappings for each set of matched vertices in G_i to a coarse vertex in G_{i+1}. Note that, following each coarsening pass, both of these arrays are retained in memory to hold the data required during un-coarsening phase. Although the use of *Mtchs* sounds redundant, knowing that the pair of matched vertices are linked together via p fields in the matched vertices, it is in fact essential so as to be able to perform prefix-sum operation which operate on array data structure. Therefore, having the *Mtchs* populated, similar to mt-metis, the prefix-sum based algorithm computes the mappings from matched vertices in G_i to coarse vertices in the G_{i+1}. This algorithm is fully described in section 4.2.1 using an example.

Upon completion of the matching, the contraction stage starts by receiving the mappings stored in *CMaps* in order to build the CSR data structure (*NbrVrtxIndices*, *EdgeWeights,* and *Vrtcs* arrays) of next level coarser graph G_{i+1}. In doing so, we require some temporary tables to perform segmented scan operations such as prefix-sum, “sort” and “pack” operation as we explain in the following.

First, a temporary array called *VrtcsTemp* is created with its size equal to the number of vertices in G_{i+1}, and then each of its elements, which corresponds to a coarse vertex *v* of G_{i+1}, receives the sum of the number of adjacent vertices (degree) of its matched vertices V^v in G_i. Additionally, we create a temporary array called *NbrVrtxIndicesTemp* in which for each coarse vertex v the adjacency lists of vertices in V^v reside next to each other so as to form a new segment. Lastly, to store the edge weight associated to each vertex in the adjacency lists of each newly created segments, a temporary table called *EdgeWeightsTemp* is created.

Subsequently, given above temporary tables, a prefix-sum performed on *VrtcsTemp* locates the starting position within *NbrVrtxIndicesTemp* to which the adjacency lists of vertices in V^v will be stored. Recall that the segments are required to be marked using a temporary head flag array; thus, the marking is performed within the same process. Having *VrtcsTemp* populated with starting position of each segment, the *NbrVrtxIndicesTemp* is populated by the adjacency lists of all matched vertices. In doing so, each thread processing v copies the adjacency lists of vertices in V^v to *NbrVrtxIndicesTemp* while in the meantime storing the corresponding edge weight in *EdgeWeightsTemp*. Since to combine the edges in G_i we require their corresponding endpoints in G_{i+1}, the *NbrVrtxIndicesTemp* will be populated by the indices of coarse vertices. Note that this process is performed fully in parallel with no data sharing between threads whatsoever.

Next, the "sort" operation is performed on *NbrVrtxIndicesTemp* wherein the indices of coarse vertices in each segment represent the sort keys. Upon completion of the "sort", in each segment the vertices with common coarse vertices reside next to each other. Successively, utilizing the "warp segmentation" technique, in each segment one of the duplicate vertices are marked to be removed, while the sum of the edge weights is stored in corresponding element in *EdgeWeightsTemp*. Moreover, if the vertex of G_{i+1} in the segment is equal to the corresponding coarse vertex of the belonging vertex, that is, an edge exists between two matched vertices, the vertex is marked so as to be removed. The result of such marking is stored in a temporary head flag array *FlagTemp*, wherein the *0* values signify the vertex to be removed and *1* values imply the remaining ones with the aggregate edge weights.

Having the *FlagTemp* populated with markings, the "pack" operation is performed to build the *NbrVrtxIndices* and *EdgeWeights* arrays of the next level coarser graph G_{i+1}. By performing the "pack" operation the index offset of each remaining vertex within its adjacency list in G_{i+1} is obtained. Subsequently, having the result of "pack" operation, the size of adjacency list corresponding to each coarse vertex v in G_{i+1} is found by incrementing the last index offset of each segment by one, and stored in a temporary array *SizeTemp* ,with its size equal to number of vertices in G_{i+1}, in a corresponding location to *v*. A prefix-sum operation on *SizeTemp* results in the offset address of each adjacency list within *NbrVrtxIndices* of G_{i+1}; the result of this operation is stored in temporary array *OffsetTemp*. Having the address within *NbrVrtxIndices* for each adjacency list (stored in *SizeTemp*) together with the index offset for each adjacent vertex within it (stored in *OffsetTemp*) the vertices corresponding to the remaining edges for each *v* is copied from its adjacency list segment in *NbrVrtxIndicesTemp* to *NbrVrtxIndices* of G_{i+1}. Finally, the *Vrtcs* of G_{i+1} is constructed by populating the value of *noOfEdges* and *indOfEdges* for each *v* from *SizeTemp* and *OffsetTemp*, respectively. Then the *v*'s weight in w field is set via the *CMaps* and *Mtchs* arrays.

Cost Analysis

Given a GPU-like machine with *p* processors in the "*Threaded Many-core Memory*" (*TMM*) model [39], [40], the time complexity (T_p) of an algorithm is comprised of both the computational and memory complexity; it is assumed that the program is perfectly scheduled. The computational complexity is formulated in terms of work T_1, the total amount of computation steps using one processor, and span T_∞, the amount of computation steps on the critical path wherein the number of processors are presumed to be infinite. The memory complexity is derived using a parameter μ which is the total number of memory transfers (either grouped or not) from/to global memory to/from shared memory. Moreover, T_p depends on a number of machine/program parameters such as the latency to slow memory *L*, memory bandwidth to slow memory *C*, hardware limit on the number of threads per core *X*, size of fast local memory per core group *Z* and the number of available hardware threads τ:

$$T_p = O(max\ (T_1/p,\ T_\infty,\ \mu L\ /\ \tau p))$$

The following is the summary of major steps in each pass of the coarsening phase we use as quick reference during the computational and memory analysis utilizing the (TMM) model:

1. Given graph G_i, for each vertex find a match candidate utilizing the HEM technique and the "warp segmentation" along with the internal parallel reduction (max) function.
2. Execute STM transactions to perform the actual matching using the candidates stored in vertices to finalize the match.
3. Populate the *Mtchs* array by fetching the result of matching stored in the vertices.

4. Perform the steps including the prefix-sum operation on *Mtchs* to compute/store the mappings in *CMaps*.
5. Populate each element of *VrtcsTemp* with the total number of adjacent vertices of the matched vertices.
6. Perform prefix-sum on *VrtcsTemp* to find the positions of the adjacency lists of each coarse vertex in *NbrVrtxIndicesTemp/ EdgeWeights* before merging.
7. "Copy" the adjacency lists from *NbrVrtxIndices* / *EdgeWeights* to *NbrVrtxIndicesTemp* / *EdgeWeightsTemp*.
8. Perform "sort" operation on adjacency list as segments in NbrVrtxIndicesTemp.
9. Using the "warp segmentation", mark the duplicate vertices to be removed in each segment of *NbrVrtxIndicesTemp*, while storing the aggregate edge weights of the corresponding edges in *EdgeWeightsTemp*.
10. Perform "pack" operation to obtain the index offset of each coarse vertex *v* within its adjacency list of *NbrVrtxIndices* in G_{i+1}.
11. Populate the *SizeTemp* with the size of adjacency list corresponding to each *v*.
12. Perform prefix-sum operation on *SizeTemp* to find each *v*'s adjacency list starting address within *NbrVrtxIndices* in G_{i+1}, storing the result in *OffsetTemp*.
13. Using *SizeTemp* and *OffsetTemp*, "copy" the remaining vertices (after removing the duplicates) from adjacency list segments of *NbrVrtxIndicesTemp* to *NbrVrtxIndices*.
14. Populate the elements of *Vrtcs* in G_{i+1} using *SizeTemp*, *OffsetTemp*, *CMaps* and *Mtchs* arrays.

From the above list, five major categories of operations can be inferred: the parallel scan (both segmented and un-segmented versions), "warp segmentation", "copy", "populate" and matching STM transaction. Using TMM model, our approach to analyze the time complexity is to find the largest term among the cost of these categories for T_1, T_∞ and μ parameters required to come up with T_p equation. Note that the input graph is assumed to be connected with n vertices and m edges, i.e., $m>=n$.

The scan and segmented scan primitives as well as operations based on them, as discussed earlier in section 4.3, has the total work of $T_1 = O(n)$ and the span of $T_\infty = O(log_2(n))$, and since the memory access patterns in these primitives are regular, and hence, predictable, the memory transfers can be grouped in chunks of size C, resulting in $\mu = O(n/C)$.

The steps using the "warp segmentation" involve processing all the edges which results in the work $T_1=O(m)$ and the span $T_\infty =O(m/n)$. Using "warp segmentation" technique, the edges are processed in chucks of size C, therefore, the total number of memory transfers is $\mu = O\ (m/C)$.

For the "copy" operation, it is easy to show that total work is $T_1 = O(m)$ and the span is $T_\infty =O(1)$ while the total number of memory transfers is $\mu = O(m/C)$ by grouping the memory accesses. Similarly, In the "populate" operation, the total work is $T_1 = O(n)$, while the span is $T_\infty = O(1)$; however, the memory accesses are not predictable so that it incurs $\mu = O(n)$ of memory transfers.

The STM transactions has the total work of $T_1 = O(n)$ when one processor performs the transactions serially, i.e., is absence of synchronization. The total number of memory operations is $\mu = O(m)$ as the memory accesses are not predictable. The span is $T_\infty = O(\Delta)$, i.e., the maximum degree of graph. This span ($O(\Delta)$) indicates the upper bound of the total time required when all the threads, each processing one of the adjacent vertices of a common vertex with degree Δ, finish their transactions such that only one of them can succeed to match; recall that the number of p is assumed infinity in T_∞. With similar logic, the total number of passes in the coarsening phase (which is equal to that of un-coarsening phase) is $O(\Delta)$ because in each pass STM transactions match and contract one edge out of all the remaining edges with common vertex. Note that threads run concurrently when executing these transactions on all the vertices of graph. As a result, the runtime T_p equation can be expressed as the following:

$$T_p = \Delta\ O(max\ (m/p,\ m/n,\ \Delta,\ log_2\ (n),\ m\ L\ /\ \tau p))$$

In the above equation, the first term is $O(m/p)$, since in both "warp segmentation" and "copy" the total work is $T_1 = O(m)$; note that we assumed $m>=n$. Also depending on the number of vertices/edges in the input graph and graph's maximum degree Δ, the span T_∞ is the maximum of $O(log_2(n))$, due to the cost of scan primitive; $O(\Delta)$, due to STM transactions; and $O(m/n)$ due to the "warp segmentation". Finally, the total number of memory operations is the largest term $O(m)$ which belongs to the STM transactions.

4.4.3 Un-coarsening

Given a coarse graph G_i, each pass of un-coarsening phase consists of projection stage followed by multiple iterations of refinement stage until there is no substantial improvement can be made in the edgecut. That is, the partition P_i obtained for G_i, is projected to finer graph G_{i-1} where $i>=1$ producing initial version of P_{i-1}, and then, P_{i-1}'s edgecut is iteratively improved by the vertex moves followed by updating the gain values for boundary vertices affected by the changes in the edgecut. This process continues until the edgecut is no longer getting any significant improvement. The un-coarsening phase ends when the partition projected to the original graph is refined, leading to the desired partition, i.e., the result of the graph partitioning.

Parallelization of projection stage is straightforward. It is carried out by mapping the partition's part label (number) assigned to each coarse vertex v: $P_i[v]$ to the set of matched vertices V^v in next level finer graphs G_{i-1}. In particular, each thread processing a vertex u of G_{i-1} belonged to V^v first finds the coarse vertex v in G_i which is mapped to, and then sets u's partition's part number to that of v, that is:

$$P_{i-1}[u] = P_i[v] : i>=1$$

Recall that the refinement stage involves the moving of the boundary vertices to adjacent parts in the current partition with the purpose of reducing the edgecut. However, to obtain the order of these vertex moves, the boundary vertices are required to be prioritized based on their gains. Thus, we require to initally compute the gain for each boundary vertex before performing the actual vertex moves.

Given a partition, the gain of a boundary vertex v is equal to the maximum value among all the edgecut reduction made against the external parts, assuming the move is individually performed to such parts. That is, assuming v is moved to the external partition's parts one at a time, the maximum reduction in edgecut that can be realized among all these moves is the gain associated to v. As it will be explained with more detail in the following, using the segmented scan based operations ("sort" and prefix-sum) and the "warp segmentation", the v's adjacency list are grouped into segments based on the common partition's parts at their endpoints and then the per-part edgecut is computed so as to calculate the gain associated to v.

Following the actual move made for each vertex *v*, the vertices adjacent to *v* need to reflect the changes in their gain values. Therefore, during each vertex move, the boundary vertices adjacent to *v* are marked as having invalid gains so that they are not considered to be moved in the current iteration, requiring their gain to be computed again in the following iteration. That is, each iteration alternates between gain calculation and vertex moves. Moreover, after *v* moved to other part of the partition, it is marked as inactive to be excluded from further moves in current refinement stage. This helps expedite the convergence of overall refinement process. Note that when vertices are marked as inactive, they don't require gain recalculation any longer, therefore, helping to performance improvement by reducing the unnecessary computation.

Recall that in mt-metis, to leverage the concurrent processing using multiple cores, the computation is relaxed by using per-thread priority queues rather than a single, global queue to insert vertices based on their gains. Consequently, the vertex with highest grain can be extracted by each thread from its dedicated queue before performing the vertex moves. This approach, however, is not appropriate for the GPU architecture mainly due to high number of hardware threads.

To streamline the refinement process while increasing the parallelism by leveraging thousands of GPU cores, we apply a heuristic to approximate the behavior of the priority queues used in mt-metis. In doing that, we iteratively select and move vertices whose gains fall into the given range of gain values starting from the range which contains highest values. To allow the chance of move to boundary vertices, at each iteration of refinement the given range is extended to cover lower (positive) gain values. The second advantage of this approach, other than being suited for GPU, is to reduce the contention among STM transactions by reducing the number of threads which process the transactions.

As briefly mention earlier, we use the segmented scan based operations to find the gain for each boundary vertex together with the partition's part against which such gain is achieved. First, the adjacency lists stored in *NbrVrtxIndices* are sorted using the partition's part number as sort key. This is done via the "sort" operation on adjacency lists array *NbrVrtxIndices* wherein each adjacency list represents a segment. Then, using the "warp segmentation" the new, smaller segments containing the vertices with same partition's part number are marked within each adjacency list. This results in a segmented array used as the input to the segmented prefix-sum operation to compute the per-part edgecut for each boundary vertex v.

Recall that to compute the gain we require to compute the total weights of edges connecting *v* to the adjacent vertices residing in the same partition's part to be subtracted from the v's maximum per-part edgecut. As such, , given the result of the prefix-sum operation, *v*'s gain is computed using the "warp segmentation" with the parallel reduction (*max*) by first finding the maximum per-part edgecut to external partition's parts, and then subtracting it by the total weight of edges connected to the partition's part *v* resides in. Note that here we compute the exact value of v's gain as opposed to the estimated gain computed in mt-metis.

To move a boundary vertex and marking its boundary adjacent vertices indicating that their gains are not valid for current iteration, we use STM transactions ensuring the consistency of such refinement-related data. That is, using STM transactions guarantees the boundary vertices in source and destination partition's parts are not moved by other threads' transactions during vertex move. In addition, during STM transaction we need to inactivate the boundary vertex to avoid further move in current refinement iteration and updates its partition's part number. Lastly, to ensure the partition balance is maintained during refinement, extra checks are required within STM transaction before updating the weight of partition's parts (both the source and destination parts) involved in a vertex move.

Since the number of parts in a given partition is significantly smaller than the number of vertices in a large graph, simply using the general STM transaction on the involved partition's parts will lead to the bottleneck and overall poor performance of the graph partitioning. This issue can be addressed by increasing the throughput of transactions to mitigate the side effect of such bottleneck. In doing so, we employ an *inner transaction* which takes care of update to the source and destination parts, within the main (outer) transaction; when both inner and outer transactions performed successfully the vertex move is complete. That is, the thread performing the inner transaction attempts to update the weights of the partition's parts affected by a tentative vertex move. When an attempt failed, the inner transaction is rolled back and restarted until it commits; such commit operation also causes the outer transaction to commit. However, these attempts are performed as long as a balanced partition is still attainable, otherwise both the outer and inner transactions are rolled back with no further attempt.

Cost Analysis

Similar to the complexity analysis of the coarsening phase, we list the major steps of each pass during the un-coarsening phase, followed by categorizing these steps to simplify the analysis process to find the time complexity T_p:

1. Perform projection in which each thread processing a vertex in finer graph G_{i-1} populates its partition's part using that of corresponding coarser vertex in G_i.
2. Perform segmented based "sort" operation on *NbrVrtxIndices* to put the vertices with the same partition's part next to each other in each adjacency list.
3. Using the "warp segmentation", create the new smaller segments within each adjacency list where the vertices with the same partition's part.
4. Perform segmented prefix-sum to compute the per-part edgecut each boundary vertex v, including to the partition's part v is residing in.
5. Set the gain range in the CPU host if this is the first time or extend it to cover lower gain values.
6. Find the v's gain by performing parallel reduction using the "warp segmentation" to find the maximum of the total edge weights to external partition's parts, and then subtracting it by the total weight of edges connected to the partition's part v currently resides in.
7. Perform STM transactions to move each boundary vertex which its gain falls into the given range of gain values.
8. Repeat starting from step 2 until the number of moves falls below some threshold or there is no boundary vertex with positive gain left.

Examining the above steps, the following four major categories of operations can be extracted: the segmented scan ("sort" and prefix-sum), "warp segmentation", "populate", and the STM transactions. Similar to the analysis we made in coarsening phase, we find the largest terms for T_1, $T\infty$ and μ parameters among these categories in order to find T_p.

As explained in coarsening phase, recall that an segmented scan based operation has the total work of $T_1 = O(n)$, the span of $T\infty = O(log_2(n))$, and total number of memory transfers $\mu = O(n/C)$. Using the "warp segmentation" results in work $T_1 = O(m)$, span $T\infty = O(m/n)$, and the total number of memory transfers is $\mu = O(m/C)$. For the "populate" operation, the total work is $T_1 = O(n)$ and the span is $T_\infty = O(1)$ while the total number of memory transfers is $\mu = O(n)$.

The STM transactions has the total work of $T_1 = O(m)$ where we assume one processor performing the STM transactions serially whereas no synchronization required. To find the span, we closely study the bottleneck of the process which is the most expensive item wherein the inner transactions acquire the partition's part's locks when moving vertices from one part to the other. We consider a condition where each thread has to wait for all the other threads in other parts to finish their transactions when moving a vertex to one of the other (k-1) parts. As a result, such thread needs to wait maximum O(k) times to perform its transaction. Moreover, there are maximum $n/(k(2m/n))$ waiting (outer) transactions in each part where ($2m/n$) is average degree of each vertex, i.e., $T_\infty = O(k\ \ (n^2/2m\ k)\ )= O(n^2/m),$ and since we assumed $m>=n$, this is results in span $T_\infty=O(n)$.

Total number of memory operations is $\mu = O(m)$. Note that the memory accesses are not predictable so that we can't group the memory transfers into chucks of C size. As discussed in coarsening phase complexity analysis, the total number of passes in un-coarsening is $O(\Delta)$. Therefore, the runtime T_p equation is expressed as the following:

$$T_p = \Delta\ O(max\ (m/p,\ n,\ m\,L\ /\ \tau\, p))$$

In the above equation, the first term is $O(m/p)$ due to the "warp segmentation" wherein the total work is $T_1 = O(m)$, assuming $m>=n$. The span T_∞ and the total number of memory operations μ are equal to $O(n)$ and $O(m)$, respectively, which both terms are due to STM transactions.

4.5 Summary

In this chapter we discussed the implementation aspects for an efficient transactional-memory-based multilevel graph partitioner on massively multithreaded GPUs in two of its phases: coarsening and un-coarsening. STM transactions were applied to matching, during coarsening, and refinement, during un-coarsening, wherein synchronized accesses to shared data are required. Additionally, we utilized “warp segmentation” to maximize throughput by reducing the load imbalance during marking processes. Whenever possible, we employed GPU specific operations aiming to improve the efficiency and scalability. In doing so, we used segmented scan primitives along with different high-level primitives derived from them, e.g. “sort” and “pack”. We concluded the chapter by analyzing the complexity of our implementation using the TMM model.

5 CONCLUSION & FUTURE WORKS

General Purpose GPUs (GPGPUs) are ideal platforms for regular computations. However, majority of real world multithreaded applications require access to shared memory with irregular access patterns. Among these irregular computations, those using morph algorithms are more challenging mainly due to unpredictability and high cost of synchronization overhead, and the complexity of code implementation. We have addressed these issues in our transactional-memory-based approach.

In our research, we chose 2 case studies of morph algorithms: the Borouvka's algorithm for calculating MSF and multilevel graph partitioning. First, we identify the major phases of the algorithm which requires synchronization to shared data. If we could extract certain algebraic properties (e.g., monotonicity, idempotency, associativity), we can use lock-free synchronizations for performance. Otherwise, we used our priority based STM for synchronization, which facilitates a natural transition from algorithm design to implementation. In addition, we employed several techniques in different phases of the algorithms, e.g., "warp segmentation" and segmented scan primitives, to achieve load balance and enhanced GPU resource utilization.

Experimental results show that our GPU-based implementation of Borouvka's algorithm outperforms both the fastest sequential implementation and the existing STM-based implementation on multicore CPUs when tested on large-scale graphs with diverse densities. Moreover, we compared the efficiency and scalability of our approach with existing approaches on GPU.

Finally, to show the applicability of our approach to other morph algorithms, we discuss a pen-and-paper design and implementation of STM-based multilevel graph partitioning and its complexity analysis using the TMM model.

In future works, we plan to apply similar transactional-memory-based approaches to other morph graph algorithms and investigate the possibility of generalizing the approach for other GPU-based irregular applications.

REFERENCES

[1] J. Hennessy, D. Patterson, “Computer architecture: a quantitative approach” Elsevier, 2011.

[2] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui, “The tao of parallelism in algorithms,” In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, pp. 12–25, ACM, 2011.

[3] O. Borůvka, “O jistém problému minimálním (About a certain minimal problem),” Práce mor. přírodověd. spol. v Brně III 3, pp. 37-58, 1926.

[4] V. Vineet, P. Harish, S. Patidar, P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” In Proceeding of the Conference on High Performance Graphics, pp. 167-171, ACM, 2009.

[5] R. Nasre, M. Burtscher, K. Pingali, “Morph algorithms on GPUs,” In Proceedings of ACM SIGPLAN Notices, 48(8), pp. 147-156, ACM, 2013.

[6] R. Nasre, M. Burtscher, K. Pingali, “Atomic-free irregular computations on GPUs,” In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pp. 96-107, ACM, 2013.

[7] N. Shavit, D. Touitou, “Software transactional memory”, Distributed Computing, 10(2), pp. 99-116, 1997.

[8] D. Cederman, P. Tsigas, M.T. Chaudhry, “Towards a software transactional memory for graphics processors,” In Proceedings of Euro-graphics Symposium on Parallel Graphics and Visualization, pp. 121-129, 2010.

[9] Y. Xu, R. Wang, N. Goswami, T. Li, L, Gao, D. Qian, ”Software transactional memory for GPU architectures,” In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, 2014.

[10] A. Holey, A. Zhai, “Lightweight software transactions on GPUs, ” In Proceedings of 43rd International Conference on Parallel Processing, pp. 461-470, IEEE, 2014.

[11] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, G. Morgan, “PR-STM: Priority Rule Based Software Transactions for the GPU,” European Conference on Parallel Processing, pp. 361–372, 2015.

[12] F. Khorasani, R. Gupta Laxmi, N. Bhuyan, “Scalable SIMD-Efficient Graph Processing on GPUs,” In Proceedings of International Conference on Parallel Architecture and Compilation, pp. 39-50, IEEE, 2015.

[13] Nvidia. CUDA. Retrieved September 7, 2016 from http://www.nvidia.com/cuda.

[14] L. G. Valiant. “A bridging model for parallel computation,” Communications of the ACM, 33(8), pp. 103-111, 1990.

[15] M. Herlihy, J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” In Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289-300, ACM, 1993.

[16] C. J. Rossbach, O. S. Hofmann, E. Witchel, “Is transactional programming actually easier?” In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 45(5), pp. 47-56, 2010.

[17] V. Pankratius, A.-R. Adl-Tabatabai, “A study of transactional memory vs. locks in practice,” In Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp 43-52, ACM, 2011.

[18] R. Ennals, “Efficient software transactional memory,” Technical report, Intel Research Cambridge, UK, 2005.

[19] T. Harris, J. Larus, R. Rajwar, “Transactional Memory,” Synthesis Lectures on Computer Architecture, 5(1), pp. 1-263, 2010.

[20] T. Harris, K. Fraser, “Language Support for Lightweight Transactions,” In ACM SIGPLAN Notices, 38(11), pp. 388-402, ACM, 2003.

[21] Y.C. Tseng, T.T.Y. Juang, M.C. Du, “Building a multicasting tree in a high-speed network,” IEEE Concurrency, 6(4), pp. 57-67, 1998.

[22] L. An, Q.S. Xiang, S. Chavez, “A fast implementation of the minimum spanning tree method for phase unwrapping,” IEEE Transactions on Medical Imaging, 19(8), pp. 805-808, 2000.

[23] S. Kang, D.A. Bader, “An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs,” In Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, 44(4), pp. 15-24, 2009.

[24] S.V. Adve, K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, IEEE Computer Journal 29(12), pp. 66-76, 1995.

[25] DIMACS 10 challenge graph collection – Graph Partitioning and Graph Clustering, Retrieved January 25, 2016 from http://www.cc.gatech.edu/dimacs10/downloads.shtml.

[26] D. A. Bader, K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," Technical Report, 2006.

[27] J. Siek, L. Lee, A. Lumsdaine, "The Boost Graph Library: User Guide and Reference Manual," Addison-Wesley, 2002.

[28] D. Lasalle , G. Karypis, “Multi-threaded Graph Partitioning,” In Proceedings of 27th International Symposium on Parallel and Distributed Processing, pp. 225-236, IEEE, 2013.

[29] T. Bui, C. Jones, “A heuristic for reducing fill in sparse matrix factorization”, In Proceedings of the 6th SIAM Conference, Parallel Processing for Scientific Computing, pp. 445-452, 1993.

[30] R. Leland, B. Hendrickson, “A Multilevel Algorithm for Partitioning Graphs, Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[31] G. Karypis, V. Kumar, “Analysis of Multilevel Graph Partitioning,” Technical Report, TR 95-037, Department of Computer Science, University of Minnesota, 1995.

[32] G. Karypis, V. Kumar, “A fast and highly quality multilevel scheme for partitioning irregular graphs”, SIAM Journal on Scientific Computing, pp. 359-392, 1998.

[33] B. W. Kernighan, S. Lin, “An efficient heuristic procedure for partitioning graphs”, Bell System. Technical Journal, 49, pp. 291-307, 1970.

[34] C. M. Fiduccia, R. M. Mattheyses, “A linear time heuristic for improving network partitions”, In Proceedings of 19th IEEE Design Automation Conference, pp. 175-181, 1982.

[35] D. LaSalle, G Karypis, “A parallel hill-climbing refinement algorithm for graph partitioning,” 45th International Conference on Parallel Processing, pp. 236-241,IEEE, 2016.

[36] M. Harris, S. Sengupta, J.D. Owens, “Parallel prefix sum (scan) with CUDA,” GPU gems, 3(39), pp. 851-876, 2007.

[37] J.T., Schwartz, “Ultracomputers,” ACM Transactions on Programming Languages and Systems , 2(4), pp. 484-521, 1980.

[38] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, "Scan primitives for GPU computing," In Graphics hardware, vol. 2017, pp. 97-106, 2007.

[39] L. Ma, K. Agrawal, R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," Future Generation Computer Systems, vol. 30, pp. 202-215, 2014.

[40] M. Lin, R.D. Chamberlain, K. Agrawal, "Analysis of classic algorithms on GPUs," In Proceedings of High Performance Computing & Simulation, pp. 65-73, IEEE, 2014.

[41] B. Goodarzi, M. Burtscher, D. Goswami, "Parallel Graph Partitioning on a CPU-GPU Architecture," In Proceedings of Parallel and Distributed Processing Symposium Workshops, pp. 58-66. IEEE, 2016.