CONCURRENT AND PARALLEL SYSTEMS: ROBUST ENHANCEMENT OF PERFORMANCE

by

ANSHU S ANAND
ENGG01201104011

Bhabha Atomic Research Centre, Mumbai

A thesis submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

for the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

February, 2019

Homi Bhabha National Institute[1]

Recommendations of the Viva Voce Committee

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by Mr. Anshu S. Anand entitled "Concurrent and Parallel Systems: Robust Enhancement of Performance" and recommend that it may be accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

[illegible] 08/2/19

Chairman - Prof. A. P. Tiwari Date:

[illegible] 08/02/2019

Guide / Convener - Prof. A. K. Bhattacharjee Date:

[illegible] 8/2/19

Co-guide - Prof. R. K. Shyamasundar Date:

Shivali 08/02/19

Examiner - Dr. Shivali Agarwal Date:

V.H. Patankar 08.02.2019

Member 1- Prof. V. H. Patankar Date:

Member 2- Prof. S. Kar Date: 08.02.2019

Skar

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

We hereby certify that we have read this thesis prepared under our direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: 05/03/2019

Place: Mumbai

[illegible] [illegible]

Co-guide **Guide**

STATEMENT BY AUTHOR

ANSHU S ANAND

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

ANSHU S ANAND

List of Publications based on this thesis

Journal Publications

1. "STMs in Practice: Partial Rollback vs Pure Abort Mechanisms", Anshu S Anand, R. K. Shyamasundar, Sathya Peri, Concurrency and Computation: Practice and Experience, Wiley, 2018, DOI:10.1002/cpe.4465.

2. "A Deadlock-free lock-based synchronization for GPUs", Anshu S Anand, Akash Srivastava, R. K. Shyamasundar, Concurrency and Computation: Practice and Experience, Wiley, 2018, DOI:10.1002/cpe.4991.

Communicated

1. "Software Transactional Memory as a parallel Programming Paradigm", Anshu S Anand, R. K. Shyamasundar and Anup K. Bhattacharjee, communicated and under review in BARC Newsletter.

Conference Publications

1. "Opacity proof for CaPR+ algorithm", Anshu S Anand, R. K. Shyamasundar, Sathya Peri, In proceedings of the 17th international conference on distributed computing and networking, ICDCN '16, Singapore, ACM, 2016, DOI: 10.1145/2833312.2833445.

2. "Scaling Computation on GPU Using Powerlists", Anshu S Anand, R. K. Shyamasundar, In proceedings of the IEEE 22nd International Conference on High Performance Computing Workshops (HiPCW), Bangalore, IEEE, 2015, DOI: 10.1109/HiPCW.2015.14.

Anshu S. Anand

DEDICATED

to

THE ALMIGHTY GOD

ACKNOWLEDGEMENTS

First and foremost, I would like to express deepest gratitude to my guide Prof. R. K. Shyamasundar for the guidance and support at every stage of my research work. His enthusiasm and passion for research has been an inspiration and has played an instrumental role in keeping me motivated. He has always made himself available, even when he had been traveling, through emails. His encouragement, advices and suggestions throughout this period, especially during the revision of papers, have helped me evolve as a better researcher.

I would also like to express sincere gratitude to my supervisor Dr. A. K. Bhattacharjee for his guidance and encouragement during this time. His friendly attitude helped me confide in him my problems, and his advices have indeed helped me sail through. I thank him for his support throughout in spite of his busy schedule.

I thank my doctoral committee chairman Prof. A.P. Tiwari for his continuous support and valuable suggestions during my research work and for monitoring my progress every year. I would like to express my sincere thanks to the doctoral committee members for their critical comments and suggestions during annual review presentations.

I thank Prof. Sathya Peri, our collaborator from IIT Hyderabad for his significant contributions in this research work. The numerous brainstorming sessions and discussions with him and his students were very helpful and has contributed a lot in this work. I would also like to thank Dr. Anjil Srivastava, Dr. N V Narendra Kumar and Dr. Vishwas Patil for being a mentor and friend alike. Their advices, feedbacks and constructive criticisms have played a vital role in this long journey. I also thank Akash for his contributions to this work.

I would also like to thank TIFR and IIT for providing the necessary infrastructure for me to carry out this work, and to the Computer Centre, BARC and CUDA Centre of Excellence, IIT Bombay for providing access to their computational facilities to carry out the experiments. Special thanks to Mr. Ravikumar, TIFR for facilitating access to the Tilera machines at TIFR, and for being a good friend. I am indebted to Homi Bhabha National Institute, Mumbai for giving me this opportunity to carry out the research work, and to Department of Atomic Energy (DAE) for providing me the generous DAE Graduate Fellowship.

I would like to thank my friends and colleagues at TIFR and IIT, especially Namrata, Tejas, Deepthi and Nikita for their moral support and company. I want to give special thanks to Namrata and Vatsala Aunty for making my stay at Mumbai as comfortable as never before.

I extend my heart-felt thanks to my parents, family and friends for their love, care and support. In addition, I want to thank my wife, Ashwini for her patience in this entire endeavor, who has sacrificed so much.

Above all, I express my indebtedness to the “ALMIGHTY” God, for all his blessings and kindness, without which this work would not have been performed.

CONTENTS

		PaPage No.
SYNOPSIS		i
LIST OF FIGURES		ii
LIST OF TABLES		iv
CHAPTER 1	**INTRODUCTION**	1
	1.1. Objective	8
	1.2. Organization of the thesis	10
CHAPTER 2	**A PERFORMANCE EFFICIENT SOFTWARE TRANS-ACTIONAL MEMORY**	11
	2.1. Software Transactional Memory	12
	2.2.1. Literature Survey	12
	2.2.2. System Model	20
	2.2. Enhanced Automatic Checkpointing and Partial Rollback (CaPR+) Algorithm	23
	2.2.1. Data Structures used in CaPR+ Algorithm	25
	2.2.2. Description of CaPR+ Algorithm	27
	2.2.3. Correctness of Transactional Memories	35
CHAPTER 3	**COMPARATIVE PERFORMANCE EVALUATION OF CAPR+ ALGORITHM**	45
	3.1. Benchmarks for STMs	45
	3.2. Performance Evaluation of CaPR+ Algorithm	48
	3.2.1. Comparative Evaluation of Abort and Partial Rollback mechanisms	49
	3.2.2. Comparative Performance of CaPR+ with CaPR+ (abort)	53
	3.2.3. Comparative performance of CaPR+ with RSTM	55
	3.3. Integrated Partial Rollback-Abort Framework	57
	3.4 Discussion	59
CHAPTER 4	**A DEADLOCK-FREE LOCK-BASED SYNCHRONIZA-TION FOR GPUS**	61
	4.1. Background	64
	4.1.1. Overview of GPU architecture	64
	4.1.2. CUDA programming model	65
	4.2. Deadlocks	67
	4.2.1. Classic Deadlocks	67
	4.2.2. Deadlock Scenarios in GPUs	69

4.2.3. Overcoming Deadlocks in GPUs 75
4.3. Modelling Circular Deadlocks 79
4.4. Deadlock-free lock algorithms for GPUs 82
4.4.1. Prev-Deadlockfree: A Naive Deadlock Prevention Algorithm 84
4.4.2. Par-Deadlockfree: An efficient Deadlock Prevention Algorithm 87
4.4.3. Proof of Deadlock Freedom 93
4.5. Performance Evaluation 94
4.5.1. Prev-Deadlockfree vs Par-Deadlockfree 95
4.5.2. Comparative Evaluation of Par-Deadlockfree with existing works 96
4.6. Discussion 98

CHAPTER 5 EXPLOITING PARALLELISM AND RECURSION TO REALIZE PERFORMANCE 99
5.1. Background 102
5.1.1. Characteristics of GPU 102
5.1.2. Powerlists 102
5.2. Specifying Matrix Multiplication using Powerlists 105
5.3. Realizing Matrix Multiplication keeping in view the GPU Parameters 109
5.3.1. Matrix Multiplication Algorithm: An Illustrative Example 110
5.3.2 Realizing the Matrix Multiplication Algorithm 112
5.4. Scaling Matrix Multiplication over a Cluster of GPUs 123
5.4.1. Matrix Multiplication : An Illustration 125
5.4.2. Computing matrix multiplication over GPUs 126
5.4.3. Method of computing large matrices over GPU cluster using cuBLAS 127
5.4.4. Experimental Results 133
5.5. Powerlists and Oblivious algorithms 145
5.5.1. Cache-Oblivious Algorithms 146
5.5.2. Multi-core Oblivious Algorithms 148
5.6 . Discussion 153

CHAPTER 6 CONCLUSIONS AND FUTURE SCOPE 155

REFERENCES 159

SYNOPSIS

The ever-increasing need for computational power for solving large problems has led to the design of many concurrent and parallel systems. The largest of High Performance Computing (HPC) systems today are comprised of thousands of multicore processors and accelerators/special purpose hardware. However, the parallel programming languages have not been able to address the challenges brought about by the increased level of parallelism made available by the hardware. In this work, solutions have been provided to problems pertaining to the Performance, Productivity and Robustness challenges of concurrent and parallel systems.

In the context of concurrent systems, an efficient Software Transactional Memory (STM) has been proposed, which would provide a good way to address both application performance and productivity. In the context of parallel systems, powerlist data structure has been used to provide a high-level framework for performance and productivity. Powerlist is a robust data-structure that allows us to utilize recursion and parallelism in an unified manner. Further, to address robustness concerns, the variety of deadlocks possible in GPU applications has been discussed and two lock-based deadlock-free synchronization mechanisms have been presented: Prev-Deadlockfree and Par-Deadlockfree, for GPU architectures that overcomes these issues. An STM framework for GPUs has been proposed by us that internally uses the deadlock-free locks. This framework is expected to simultaneously address all the three challenges of multi-core programming, namely performance, productivity and robustness, thereby widening the scope of GPGPU computing.

LIST OF FIGURES

		Page No.
2-1	Pictorial representation of History H1	25
2-2	Local workspace of T1 and the global workspace	34
2-3	Pictorial representation of the modified History H2	42
3-1	Execution time for STAMP benchmark applications with CaPR+ and CaPR+(Abort)	51
3-2	Speedup of CaPR+ Algorithm w.r.to CaPR+(Abort)	54
3-3	Speedup of CaPR+ Algorithm w.r.to RSTM	56
3-4	Speedup of the hybrid implementation w.r.to CaPR+ , CaPR+(Abort), and RSTM	58
4-1	The GPU architecture and CUDA programming model	67
4-2	Resource allocation graphs: (a) Holding a resource. (b) Requesting a resource. (c) Deadlock	69
4-3	Shared Memory lock unit	73
4-4	Example to illustrate lock stealing	78
4-5	Example to illustrate the problem with lock stealing	78
4-6	Extended resource allocation graphs: (a) Holding a resource. (b) Requesting a resource. (c) SIMD Deadlock	80
4-7	Extended resource allocation graph of circular deadlock (at Line 3 of Table 4.2)	81
4-8	Extended resource allocation graph of circular deadlock (at Line 7 of Table 4.2)	81
4-9	Extended resource allocation graphs of circular deadlock with threads from different warps	82
4-10	Comparative evaluation of Prev-Deadlockfree and Par-Deadlockfree	96
4-11	Comparative evaluation of DMR implementations	97
5-1	Cooley-Tukey algorithm for computing FFT	100
5-2	Code snippet of kernel invocation	117
5-3	Code snippet of the matrix multiplication kernel	118
5-4	GPU Memory Hierarchy	120
5-5	Comparison of results achieved- best i Vs predicted i	122
5-6	Performance Comparison of global memory and shared memory versions with CUBLAS benchmark	122
5-7	Tree representation of the matrix partitioning	125
5-8	Execution with 4 processes, combining all the MPI transfers together, and CUDA transfers	134

5-9	Execution with 16 processes, combining all the MPI transfers together, and CUDA transfers	135
5-10	Execution with 64 processes, combining all the MPI transfers together, and CUDA transfers	136
5-11	Comparison of Total Execution times for 1, 4, 16 and 64 processes	137
5-12	Comparison of Total Execution times for 1, 4, 16, 64 processes(in log)	138
5-13	GPU implementation of radix-R Stockham algorithm	143
5-14	The HM model for 5 levels of cache	148
5-15	Multicore-oblivious matrix transposition algorithm	149
5-16	The multicore oblivious FFT algorithm	151

LIST OF TABLES

		Page No.
2.1	Properties ensured by STM protocols	18
2.2	Local Data Block	26
2.3	Shared object Store	26
2.4	Checkpoint Log	27
2.5	Global List of Active Transactions	27
2.6	Shared Memory	28
3.1	STAMP applications	47
4.1	Scheme 1	74
4.2	Example to illustrate circular deadlock within a warp	74
4.3	Example to illustrate circular deadlock across warps	75
4.4	Summary of various deadlock scenarios and their solutions	79
4.5	Example to illustrate Prev-Deadlockfree	86
4.6	Execution Time-line for the example in Table 4.5	87
4.7	Example to illustrate efficiency of Par-Sychronize1	89
4.8	Example to illustrate Par-Deadlockfree	93
5.1	GPU parameters	102
5.2	Number of blocks, block size, dimension and shared memory required at the different levels of recursion	113
5.3	Time taken for Matrix Multiplication with global memory(in ms)	119
5.4	Time taken for Matrix Multiplication with global memory(in ms)	119
5.5	Time taken for Matrix Multiplication with shared memory(in ms)	121
5.6	Time taken for Matrix Multiplication with shared memory(in ms)	121
5.7	Description of parameters used in the library functions	128
5.8	Shows the number of GPUs for which the best performance is achieved for different matrix sizes	136
5.9	Execution times of radix-R Stockham implementation	144
5.10	Execution times of cuFFT	144

Chapter 1

INTRODUCTION

Starting from the early days of programming using machine language when programs were represented by plugged interconnecting wires, the programming languages have today evolved to high-level machine-independent programming languages. However, the evolution was not smooth, and witnessed several challenges that had to be overcome [1]. We provide a brief account of the challenges faced by the programming community.

The First software crisis

The first software crisis was witnessed in 1960s and 1970s, when people were still using assembly language. The assembly languages were not portable as every hardware architecture had its own instruction set, and thus, a program had to be completely rewritten when moving to a different architecture. Also, since assembly language instructions operated at bit-level and register-level, the low level of abstraction made it very difficult to write and maintain large programs. It was felt that higher abstraction and portability was needed, without loosing performance. The solution to this crisis came in the form of high-level languages like fortran and C for Von-Neumann machines, where programmers wrote codes in the high-level language, and the compiler translated them into machine codes. A high-level language hides properties of the processor like processor registers, instruction-set and functional units, thus providing a higher level of abstraction than assembly languages. They provided programmers with a unified view of uni-processors comprising of a single flow of control and a single memory image, which considerably reduced the programmers' burden.

The second software crisis

The second software crisis was witnessed during the 80s and 90s. As computational needs increased, the software applications grew larger and larger. However, it was becoming increasingly difficult to build and maintain such complex applications with millions of lines of code, developed by hundreds of programmers. There was a need to get more composability and maintainability to ease the programmers' burden of building and maintaining large programs. This shift in the scale of applications led to increased focus on software system management. Software management needed to focus from software requirements to realization and at the same time manage migration of systems to new technology or new requirements. Object oriented programming was a solution to the second software crisis. Other solutions include better tools like Component libraries, Purify and better software engineering methodology like design patterns, specifications, testing, code reviews etc.

Performance was really not an issue, as it was believed that performance was best improved by creating faster and more efficient processors. Thus, performance was left to Moore's law which states that the number of transistors in a chip would double every 18-24 months. However, during this period, there was also considerable effort in further improving performance using parallel processing, which refers to the simultaneous use of multiple compute resources to solve a given computational problem. Many programming models have been proposed for parallel programming. Parallel programming models provide an abstract view of the hardware and memory architectures, and allow us to expose concurrency in parallel algorithms. Parallel programming models can be broadly classified on the basis of problem decomposition as follows:

1. Data parallel model

With data parallel model, same or similar computations are performed on different data repeatedly. A typical example of data parallelism is any image processing algorithm that applies a filter to each pixel of the image. OpenMP is an API that is based on compiler directives that can express a data parallel model.

2. Task parallel model

   In task parallel model, independent works are encapsulated in functions to be mapped to individual threads, which execute asynchronously. Loop iterations are a source of task parallelism, where each iteration can be considered as a separate task. Thread libraries (e.g., the Win32 thread API or POSIX threads) are designed to express task-level concurrency.

3. Hybrid models

   Sometimes, more than one model may be applied to solve one problem, resulting in a hybrid algorithm model. A database is a good example of hybrid models. Tasks like inserting records, sorting, or indexing can be expressed in a task-parallel model, while a database query uses the data-parallel model to perform the same operation on different data.

To parallelize a program, it has to be first decomposed into smaller computations that are then distributed onto different processors for execution. Two of the most common decomposition techniques used for parallelization are:

1. Functional decomposition: It is used to introduce concurrency in the problems that can be solved by different independent tasks. The computation is divided into disjoint tasks that are then run concurrently.

2. Data decomposition: It works best on an application that has a large data structure. The goal of data decomposition is to partition the data structure into multiple parts, which are then operated on by parallel tasks. The tasks performed on the data partitions are usually similar. There are different ways

to perform data partitioning: partitioning input/output data or partitioning intermediate data.

Thus, work distribution using these decomposition techniques is typically derived from task or data parallelism, which varies from problem to problem. For problems with large data structures, data decomposition may be more appropriate than functional decomposition. Task and data decomposition may also be combined to expose more parallelism. The programmer starts by using one of these patterns, and then further splits up tasks using the other pattern. A special case of such a combination is pipelining.

Thus, programmers relied on exploiting these types of parallelism in software for improving performance of sequential programs. However, the use of parallel hardware (eg. superscalar, VLIW architectures) to exploit parallelism in software does not necessarily translate into proportional performance gain. Amdahls law [2] gives a limit on the speedup that can be expected from the parallel execution of a code. It states that the performance improvement gained using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Thus, time to run the parallelized application, $T_{parallel}$ is given by

$$T_{parallel} = ((1 - P) + \frac{P}{N}) * T_{serial} + O(N) \tag{1}$$

where T_{serial}: time to run an application in serial version,

P: parallel portion of the process,

N: number of processors,

O(N): parallel overhead in using N threads.

Further, scalability remains an issue i.e. continuing to achieve this speedup with increase in the number of processors is difficult because of the following reasons:

1. there may not be enough concurrent jobs to keep the CPUs busy

2. shared work queues become a bottleneck

3. finding finer-grained parallelism is a challenge.

The development of parallel applications is really hard since parallel programming brings with it problems that were not encountered during sequential programming like deadlocks, race-conditions, non-determinism, communication, synchronization, load-balancing etc. All these issues presented some new important challenges which were not addressed by the parallel programming languages. Further, they have not been able to address the issues of natural specification of parallel algorithms and handling of the architectural features to achieve performance concurrently. In other words, there is a large gap between languages that are too low level, requiring specification of many details that obscure the meaning of the algorithm, and languages that are too high-level, making the performance implications of various constructs unclear. However, in the context of sequential computing, standard languages such as C or Java do a reasonable job of bridging such a gap. The main challenge is to bridge the gap between specification and realization (implementation) that would preserve natural specification of programs and at the same time enable realization of efficient programs on different architectures.

The third Software Crisis: 2002 -

During the start of the last decade, sequential performance was left behind by Moore's law. General-purpose unicores stopped performance scaling due to various reasons. It was becoming increasingly difficult to find any more parallelism in sequential programs, since ILP causes a super-linear increase in execution unit complexity (and associated power consumption) without linear speedup in application performance. This is referred to as ILP wall. Moreover, the sustained increase in clock frequency was primarily due to Moore's law. For almost three decades, Moore's Law was aided by Dennard scaling [3] to keep the processor power within limit. Dennard scaling law states that as the size of transistors reduces, their power density stays constant. However, due to physical limitations, it was difficult to further reduce the size of transistors and power consumption, while also increasing the operating speed [4].

Consequently, the chip operating frequency has stagnated at about 4.7 GHz. This limit on the scaling of clock speeds is referred to as Power wall. Further, the DRAM access latency is still not in line with processor speeds (referred to as Memory wall), which masks the processor speed improvements.

Due to these concerns, the performance growth in uni-processors gradually came to an end. The computer architects have eventually moved to energy-saving multi-core designs, in which multiple low-power processors are packed onto a single chip that replicate the performance of a single, faster processor. Multi-cores architectures typically aim to exploit thread-level parallelism by executing multiple threads on the different cores, thereby aiming to improve the throughput. Unfortunately, the performance of applications is not improved by merely increasing the number of cores in a chip. Ideally, upgrading to an n-core computer should result in n-fold increase in computational power. This, unfortunately, is not the case due to Amdahl's law and also because most real world computational problems cannot be efficiently parallelized without incurring the costs of the expensive inter-processor communication and synchronization operations.

Hardware has evolved at a rapid pace from uni-processors with pipelining, superscalar architecture, branch prediction, etc. to multi-core architectures to clusters. However, software has not kept the same pace. The existing parallel programming languages have not been able to address the challenges brought about by the increased level of parallelism made available by the hardware. In the current context, the main challenges of multi-core programming are as below [5]:

1. Application Performance

   To attain good application performance, the programmer needs to write the code cleverly to exploit maximum possible parallelism, while also being careful enough to get the desired (correct) results. The various factors affecting performance are:

   (a) Coverage - refers to the extent of parallelism available in an algorithm. Amdahl's law gives a limit on the potential program speedup, which is given by the fraction of code that can be parallelized.

   (b) Granularity - refers to granularity of partitioning among processors. Fine-grained parallelism offers scope for better performance as compared to coarse-grained parallelism at the cost of higher complexity. However, too small granularity may also result in high communication overhead.

   (c) Locality - programs tend to reuse data and instructions they have used recently. Temporal locality states that recently accessed memory addresses are likely to be referenced again in the near future, while spatial locality states that objects whose addresses are near one another are likely to be referenced in the near future. Thus, computations must be reordered to maximize use of data fetched for optimal performance.

2. Productivity

   Since programming for multi-cores is hard, it has been the exclusive domain of a small number of expert programmers. Moreover, achieving performance and functional correctness simultaneously can be a serious impediment to a programmer's productivity. Therefore, it is important to simplify programming to extend the domain to ordinary programmers.

3. Robustness

   Concurrent programs are hard to test and debug, due to the added complexity of expressing concurrency in programs, and also because for the same inputs, the bugs may not be reproducible in different runs. Therefore, the onus is on the programmer to write programs that are free of bugs such as data races, deadlocks, livelocks etc. In typical parallel programs, some type of coordination and exchange of data is necessary to realize the given task while also preventing occurrence of these bugs. In shared memory environments, each processor has complete access to the shared memory, so exchange of data is not required. They employ various synchronization primitives to ensure consistency of shared data and can also be used as a means to coordinate computations. In distributed systems, this is realized with communication primitives that allow sending messages and data from one node to another. Efficient synchronization primitives help in developing reliable parallel software since it allows us to structure programs as a set of synchronized operations on fine-grain objects.

1.1 Objective

In this work, we are concerned with arriving at a paradigm for enhancing performance and productivity of concurrent and parallel systems in a robust manner. In particular, we attempt to provide solutions to address the three challenges of multi-core programming discussed above:

1. Performance,
2. Productivity and
3. Robustness

In this thesis, we address the following specific problems that are part of challenges highlighted above:

1. Software Transactional Memory (STM) provides an abstraction that aids in programmer's productivity. Keeping this in view, our specific contributions are:
    (a) An improved and simplified STM algorithm for multi-threaded programs, CaPR+ has been presented that implements Partial-Rollback mechanism and proof of correctness using Opacity, a correctness criterion for STMs, has been established.
    (b) An extensive comparative evaluation of the Abort and Partial Rollback mechanisms for STMs has been carried out. Based on the findings, an integrated partial rollback-abort framework has been proposed and also implemented, that benefits from the advantages of both mechanisms. Further, several execution instances have been shown for which the hybrid

implementation outperforms all the implementations considered in our experiments. This work is the first to propose such a hybrid framework and to implement it.

2. Graphics Processing Units (GPUs) have evolved from graphics applications to general purpose applications, often referred to as GPGPU computing. Hence, it is important while transforming multi-threaded programs to GPU programs, that deadlock-free property is preserved. Towards this goal, our contributions are:

   (a) We have described possible deadlock scenarios in GPUs, and presented a formal model for SIMD and circular deadlocks in GPUs.

   (b) We have presented a novel, deadlock-free, lock-based synchronization mechanism for GPUs, establish its correctness and discuss its performance.

3. Recursion and concurrency are often considered as prime factors affecting performance. In this thesis, we use an existing robust data-structure - Powerlist [103] that allows us to utilize recursion and parallelism in an unified manner.

   (a) Matrix multiplication algorithm has been presented in a novel way, expressed in the powerlist notation, and a powerlist based approach has been presented for GPUs that predicts how threads for an application should be mapped to the GPU cores, based on the GPU parameters (eg. no. of streaming multiprocessors (SM), no. of cores per SM etc.).

   (b) The usefulness of powerlists to automatically partition the matrices has been demonstrated. In particular, we provide a library of powerlist operations that facilitate partitioning and merging of sub-problems.

   (c) A scheduling algorithm for matrix multiplication across a cluster of GPUs has been presented, thereby overcoming cuBLASs limitation to schedule an application across the cluster. An extension of this approach to Fast Fourier Transforms (FFT) has also been devised.

(d) An observation has been made that most of Cache Oblivious and Multicore-Oblivious algorithms are recursive in nature, and how they can be easily expressed using powerlists.

1.2 Organization of the thesis

The following chapters give an account of the work undertaken in this thesis. Chapter 2 provides a literature survey of Software Transactional Memory and gives the system model. Further, We give an overview of the Enhanced Automatic Checkpointing and Partial Rollback (CaPR+) algorithm, give the proof of "Opacity", a candidate correctness criterion, for the CaPR+ algorithm. In Chapter 3, we comparatively evaluate partial rollback and abort mechanisms, and present an integrated partial rollback-abort framework that exploits the advantages of both mechanisms. In Chapter 4, we present a novel deadlock-free lock-based synchronization mechanism for GPUs. In Chapter 5, we give a brief overview of the powerlist notation, present the powerlist specification for matrix multiplication and give a powerlist based approach for GPUs that predicts how threads for an application should be mapped to the GPU cores, based on the underlying hardware. Further, we discuss our method of scaling matrix multiplication and FFT over a cluster of GPUs using cuBLAS and Powerlists. Finally, we report our observation that powerlists can be used to specify most of Cache-Oblivious and Multicore-Oblivious algorithms. This is followed by conclusion in Chapter 6.

Chapter 2

A Performance Efficient Software Transactional Memory

Conventional synchronization primitives like locks and compare-and-swap (CAS) pose several challenges to the programmers and affect their productivity and/or performance significantly. Locks are difficult to manage, especially in large systems. Although coarse grained locking simplifies productivity, performance takes a hit since it limits parallelism. Whereas fine-grained locking trades off performance with productivity. Further, locks cannot be easily composed. On the other hand, CAS operates on only one word, increasing the complexity. The challenges posed by these low-level synchronization primitives led to the search of alternative parallel programming models to make the process of writing concurrent programs easier. Transactional Memory (TM) is a promising programming memory in this regard. TMs significantly improve the programmers' productivity, since they abstract out the low-level synchronization details from the user. Thus, TMs are by design productive. In this work, we optimize Software Transactional Memories (STM), a subclass of TMs, for performance. In particular, we present an optimized algorithm, Automatic Checkpointing and Partial Rollback (CaPR+) algorithm that implements partial rollback, as opposed to abort mechanism, implemented by most STMs. We discuss the proof of correctness of CaPR+ algorithm through Opacity, a popular correctness criterion for STMs. This proof of opacity has been published in [6]. We also perform an exhaustive performance evaluation of the partial rollback and abort mechanisms, which has been published in [7].

In the next section, we give a literature survey of STMs and the system model. In Section 2.2, we describe our proposed STM algorithm - *CaPR+*, and establish its proof of correctness. Finally in Section 3, we perform an exhaustive comparative evaluation of partial rollback and abort implementation of STMs through *CaPR+*, and discuss our idea of implementing an integrated abort-partial rollback framework for TMs.

2.1 Software Transactional Memory

2.1.1 Literature Survey

Transactional memory draws inspiration from database transactions. Transactions have been in use in databases successfully for decades. A database transaction specifies a semantics in which a transaction is oblivious to the presence of other transactions accessing the database and hence the programmers live in a simpler, more familiar sequential programming world. The model limits the allowable interactions between the transactions in such a way that even concurrent executions of transactions produce predictable, reproducible results. Hence, maximum parallelism is obtained if none of the transactions are trying to access the same data.

Although programming-language transactions have some similarity to the database transactions, their implementation and execution environments differ greatly, as operations in transactional databases typically involve disk accesses, whereas programs typically store data in memory. Due to this difference, it is also known as transactional memory (TM).

TM enables communication among threads running in a shared address space by executing lightweight, in-memory transactions [63].

A database transaction has four specific attributes: failure atomicity, consistency, isolation, and durability - collectively known as the ACID properties [9].

1. Atomicity: also known as all or none, it requires that all the operations in a transaction complete successfully, or that none of these actions appear to have started. If some of the operations of a transaction fail, the transaction is not allowed to finish successfully since the effects of a failed operation or transaction should not be visible to other transactions. A transaction that completes successfully commits and one that fails aborts.

2. Consistency: If a transaction modifies the state of the world, then its changes should take the system from one consistent state to another consistent state.

3. Isolation: It requires the transactions to not interfere with each other while they are running. This property makes transactions an attractive programming model for parallel computers.

4. Durability: It requires that once a transaction commits, its result is persistent and is available to all subsequent transactions.

Of these ACID properties, only the Durability property does not hold good from Transactional Memories, as unlike storage in disks, volatile memory is inherently not a durable resource.
TM provides the programmers with certain high-level constructs. With these constructs in hand, the programmer just has to identify and demarcate atomic blocks of code that should appear to execute atomically and in isolation from other threads. Also, accesses to shared memory is allowed only through the read/write transactional operations. The underlying transactional memory implementation then implicitly takes care of the correctness of concurrent accesses to the shared data. The TM might internally use fine- grained locking, or some non-blocking mechanism, but this is hidden from the programmer and the application. Thus, if the TM is implemented correctly, the programmer is less likely to introduce concurrency bugs in the code than if he or she had to handle locks explicitly. Each transaction performs operations on

shared data, and then either commits or aborts. On a successful commit, the effects of all its operations become immediately visible to other transactions; whereas in the event of an abort, all of its operations are rolled back and none of its effects are visible to other transactions. This allows the transactions to be atomic, i.e. programmers get the illusion that every transaction executes all its operations instantaneously, at some unique point in time.

Transactional memory can be implemented in hardware known as Hardware Transactional Memory [10] (HTM), in software - called Software Transactional Memory [11] (STM) or a hybrid of the two [12]. HTMs harness the power of the existing, efficient cache coherence mechanisms, as a result of which they have been found to have high performance and strong atomicity. HTM implementations typically maintain transactional meta-data in processor's cache, and the cache coherence mechanisms are employed to detect conflicts that may arise due to concurrent execution of transactions. Intel and IBM have recently introduced commodity processors with integrated HTM support [111, 112, 113, 114]. However, most of these are best effort implementations, i.e., a transaction may abort and retry because of various hardware limitations, apart from due to data conflicts as usual. Unlike HTMs, STMs allow transactions to be bounded. The main advantages of STMs are flexibility and the ease of implementation since they are implemented in software. Due to these factors, our work is based on STMs.

We now discuss the different design choices for implementing an STM.

STM design choices

Several STM implementations have been proposed, which are mainly classified based on the following metrics [9]:

1. shared object update(version management)

   It describes the mechanism by which the writes are done to the memory. There are two approaches of version management:

   (a) eager version management

      It means that the transaction directly modifies the data in memory. Hence, it is also known as direct update. The transaction maintains an undo log to hold the values written by it. If later, this transaction conflicts with another transaction, and consequently aborts, the old values held in the undo log are restored in the memory. Eager version management requires that pessimistic concurrency control be used for a transaction's writes; this is necessary because the transaction requires exclusive access to the locations if it is going to write to them directly.

   (b) lazy version management

      This approach is also known as deferred update because the updates are made to the transaction only during the commit operation of the transaction. All the updates made by the transactions are saved in a local copy of the shared object. In this way, these are non-blocking implementations. And when the transaction is about to commit, the value of the private copy of the shared object is copied into the shared object.

2. Conflict detection

   Conflict detection deals with how the conflicts are detected among different transactions. Based on this, the different strategies for conflict detection can be classified as:

   (a) Eager conflict detection

      With eager conflict detection, a conflict is detected when a transaction declares its intent to access data or at the transaction's first reference to the data, i.e. conflicts are detected as the transaction proceeds.

   (b) Lazy conflict detection

      With lazy conflict detection, conflict are detected at commit time of the transaction.

3. Concurrency Control

The three events - conflict, detection, resolution can occur at different times. Based on the times at which these events take place, there are two approaches to concurrency control:

(a) Pessimistic Concurrency Control

With pessimistic concurrency control, all three events occur at the same point in execution-when a transaction is about to access a location, the system detects a conflict, and resolves it. It allows a transaction to claim exclusive ownership of data prior to proceeding, preventing other transactions from accessing it.

(b) Optimistic concurrency control

In optimistic concurrency control, all the three events - conflict, detection and resolution can happen after a conflict occurs. It allows multiple transactions to access data concurrently and to continue running even if they conflict, as long as the TM detects and resolves these conflicts before a transaction commits.

4. Granularity

   STMs may also be distinguished as word-based or object-based, based on the granularity on which they perform logging.

5. Lock-based or Obstruction-free

   Lock-based STMs use blocking mechanisms like locks, while obstruction-free STMs do not employ blocking mechanisms, thereby guaranteeing progress even when some of the transactions are delayed.

6. Visible or Invisible reads

   A read operation performed by a transaction may be visible or invisible to other transactions.

7. Eager or Lazy acquisition of objects

   A transaction can commit only when all the objects updated by it have been acquired. In this scenario, the acquisition can take place either at the time the object is first updated (eager), or at commit time (lazy).

With visible reads, the committing transaction that updates an object can see all the reader transactions and ask them to abort. In the case of an STM with invisible reads, conflict detection is achieved through validation. A transaction needs to validate its read-set in order to ensure that the objects being accessed by it are not stale - i.e., they have not been updated by other committing transactions. In the event of a conflict, conflict resolution takes place by using the abort mechanism to undo the effects of a transaction, and re-executing it from start. Most STMs employ an abort mechanism for conflict resolution. The decision of which transaction should be allowed to proceed, and which to abort, is done by a contention manager. Several contention management policies have been devised, some of them can be found in works by Scherer et. al. [69], Guerraoui et. al. [70], and Spear et. al. [71].

Table 2.1: Properties ensured by STM protocols

System	**RSTM** [75]	**TL2** [66]	**JVSTM** [67]	**LSTM** [65]	**LSA-RT** [68]	**CaPR+**
Lock-based	no	yes	yes	yes	no	yes
Single version	yes	yes	no	yes	no	yes
Invisible reads	no	yes	yes	no	yes	no
Clock-free	yes	no	no	yes	no	yes
Opacity	yes	yes	yes	yes	yes	yes

Liveness and Progress

The different notions of liveness and progress are as given below:

1. Deadlock-free/ Non-blocking Transactions do not mutually block each other such that none of them will commit.

2. Starvation-freedom/ wait-freedom Every transaction commits infinitely often.

3. Livelock-freedom/ Progress Transaction commits occur infinitely often.

4. Obstruction-freedom If a transaction takes an infinite number of steps in isolation, it commits infinitely often.

Fair STM implementations are required to ensure the deadlock-free and starvation-free properties. Several STM implementation are available, like the Intel C++ STM compiler [13] provides language support for STM in C++, JVSTM [14] is a Java library that implements a multi-versioned STM, SwissTM [15]- a library- based STM system for C/C++ designed mainly to support large transactions, the IBM XL C/C++ compiler [16] etc.

Most of the implementations use abort mechanism for canceling the effects of operations executed within a transaction, and this incurs significant costs. Instead of re-performing the entire operations in the transaction, a partial rollback operation re-performs only a part of the operations, back to a checkpoint. This checkpoint can be set by either the programmer or the system. This may significantly reduce the overheads associated with full transaction abort.

The use of partial rollback was first illustrated by Koskinen and Herlihy in [42]. Waliullah and Stenstrom [59] suggested using checkpoints in HTMs. In the work by Lupei [43], the partial rollback operation is based only on shared data that does not support local data which requires extra effort from the programmer in ensuring consistency. Gupta et. al. [45],[44] give an STM algorithm that supports both shared and local data for partial rollback. Alice et. al. [60] give another STM that supports both shared and local data. Our work is based on that of Gupta et. al. [45].

The Automatic Checkpointing and Partial Rollback(CaPR) algorithm, is an algorithm to realize STMs. It is based on continuous conflict detection, lazy versioning with automatic checkpointing, and partial rollback. Basically, as a transaction proceeds, checkpoints are created automatically, and the changes intended on the shared objects are actually made to their local copy. As it commits, these changes are updated to the shared objects. This may lead to conflicts in other concurrent transactions. Such conflicting transactions undergo partial rollback by rolling back to the latest checkpoint(that was recorded earlier) such that they are able to observe all the updates done by the committed transactions, and then continue with their execution.

We now describe the system model in the following section.

2.1.2 System Model

The notations defined in this section have been inspired from [41]. We assume a system of n processes (or threads), $p_1, \ldots, p_n$ that access a collection of *objects* via atomic *transactions*. The processes are provided with the following *transactional operations*: *begin_tran*() operation, which invokes a new transaction and returns the *id* of the new transaction; the $write(x, v, i)$ operation that updates object x with value v for a transaction i, the $read(x)$ operation that returns a value read in x, $tryC()$ that tries to commit the transaction and returns *commit* (c for short) or *abort* (a for short), and $tryA()$ that aborts the transaction and returns A. The objects accessed by the read and write operations are called as t-objects. For the sake of presentation simplicity, we assume that the values written by all the transactions are unique.

Operations *write*, *read* and *tryC* may return a, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operation has *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations returns a or c. Abort and commit operations are called *terminal operations*. For a transaction T_k, we denote all its read operations as $Rset(T_k)$ and write operations $Wset(T_k)$. Collectively, we denote all the operations of a transaction T_i as $evts(T_k)$.

Histories. A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $evts(H)$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let $<_H$ denote the total order on the transactional operations incurred by H. With this assumption the only relevant events of a transaction T_k are of the types: $r_k(x, v)$, $r_k(x, A)$, $w_k(x, v)$, $w_k(x, v, A)$, $tryC_k(C)$ (or c_k for short), $tryC_k(A)$, $tryA_k(A)$ (or a_k for short). We identify a history H as tuple $\langle evts(H), <_H \rangle$.

Let $H|T$ denote the history consisting of events of T in H, and $H|p_i$ denote the history consisting of events of p_i in H. We only consider *well-formed* histories here, i.e., (1) each $H|T$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly *completed* with a *tryC* or *tryA* operation[a], and (2) each $H|p_i$ consists of a sequence of transactions, where no new transaction begins before the last transaction completes (commits or a aborts).

We assume that every history has an initial committed transaction T_0 that initializes all the data-objects with 0. The set of transactions that appear in H is denoted by *txns*(H). The set of committed (resp., aborted) transactions in H is denoted by *committed*(H) (resp., *aborted*(H)). The set of *incomplete* or *live* transactions in H is denoted by *incomplete*(H) (*incomplete*$(H) =$ *txns*$(H) -$ *committed*$(H) -$ *aborted*(H)).

For a history H, we construct the *completion* of H, denoted $\overline{H}$, by inserting a_k immediately after the last event of every transaction $T_k \in$ *incomplete*(H).

Transaction orders. For two transactions $T_k, T_m \in$ *txns*(H), we say that T_k *precedes* T_m in the *real-time order* of H, denote $T_k \prec_H^{RT} T_m$, if T_k is complete in H and the last event of T_k precedes the first event of T_m in H. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then T_k and T_m *overlap* in H. A history H is *t-sequential* if there are no overlapping transactions in H, i.e., every two transactions are related by the real-time order.

For two transactions T_k and T_m in *txns*(H), we say that T_k *precedes* T_m *in conflict*

[a]This restriction brings no loss of generality [61].

order, denoted $T_k \prec_H^{CO} T_m$ if: (a) (w-w order) $c_k <_H c_m$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$; (b) (w-r order) $c_k <_H r_m(x, v)$, $x \in Wset(T_k)$ and $v \neq A$; (c) (r-w order) $r_k(x, v) <_H c_m$ and $x \in Wset(T_m)$ and $v \neq A$. Thus, it can be seen that the conflict order is defined only on operations that have successfully executed.

Valid and legal histories. Let H be a history and $r_k(x, v)$ be a read operation in H. A successful read $r_k(x, v)$ (i.e $v \neq A$), is said to be *valid* if there is a transaction T_j in H that commits before r_K and $w_j(x, v)$ is in $evts(T_j)$. Formally, $\langle r_k(x, v)$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x, v)) \land (w_j(x, v) \in evts(T_j)) \land (v \neq A)\rangle$. The history H is valid if all its successful read operations are valid.

We define $r_k(x, v)$'s *lastWrite* as the latest commit event c_i such that c_i precedes $r_k(x, v)$ in H and $x \in$ *Wset*(T_i) (T_i can also be T_0). A successful read operation $r_k(x, v)$ (i.e $v \neq A$), is said to be *legal* if transaction T_i (which contains r_k's lastWrite) also writes v onto x. Formally, $\langle r_k(x, v)$ *is legal* $\Rightarrow (v \neq A) \land (H.lastWrite(r_k(x, v)) = c_i) \land (w_i(x, v) \in evts(T_i))\rangle$. The history H is legal if all its successful read operations are legal. Thus from the definitions we get that if H is legal then it is also valid.

Opacity. We say that two histories H and H' are *equivalent* if they have the same set of events. Now a history H is said to be *opaque* [50, 63] if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subset \prec_S^{RT}$. By requiring S being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted.

Implementations and Linearizations. A (STM) implementation is typically a library of functions for implementing: *read*$_k$, *write*$_k$, *tryC*$_k$ and *tryA*$_k$ for a transaction T_k. We say that an implementation M_p is correct w.r.t to a property P if all the histories generated by M_p are in P. The histories generated by an STM implementations are normally not sequential, i.e., they may have overlapping transactional operations. Since our correctness definitions are proposed for sequential histories, to reason about correctness of an implementation, we order the events in a non-concurrent history in a sequential manner. The ordering must respect the real-time ordering of the operations in the original history. In other words, if the response operation o_i occurs before the invocation operation o_j in the original history then o_i occurs before o_j in the sequential history as well. Overlapping events, i.e. events whose invocation and response events do not occur either before or after each other, can be ordered in any way.

We call such an ordering as *linearization* [46]. Now for a (non-sequential) history H generated by an implementation M, multiple such linearizations are possible. An implementation M is considered *correct* (for a given correctness property P) if every its history has a correct linearization (we say that this linearization is exported by M).

We assume that the implementation has enough information to generate an unique linearization for H to reason about its correctness. For instance, implementations that use locks for executing conflicting transactional operations, the order of access to locks by these (overlapping) operations can decide the order in obtaining the sequential history. This is true with STM systems such as [65, 64, 62] which use locks.

In the next section, we present our enhanced Automatic Checkpointing and Partial Rollback(CaPR+) Algorithm.

2.2 Enhanced Automatic Checkpointing and Partial Rollback(CaPR+) Algorithm

An STM system may be implemented as either a language construct or a library. A language construct enables compiler optimizations to improve performance and static analyses for compile-time safety guarantees, while the library approach offers greater flexibility to the language designers to experiment with various TM implementations, before infusing it into a language. In this section, we present the enhanced Automatic Checkpointing and Partial Rollback($CaPR^+$) algorithm that has been implemented as a library by us. $CaPR^+$ is an optimized version of $CaPR$ algorithm that has been modified in several aspects. In particular, most of the data structures have been simplified and the global transaction scheduler has been replaced by a policy that lets any transaction that is able to acquire locks on all objects in its write-set, to commit. $CaPR^+$ algorithm has been presented in greater detail than $CaPR$ algorithm like usage of locks and implementation details of partial rollback mechanism for better understanding and also to reason about the correctness of the algorithm. The different implementation features incorporated in $CaPR^+$ algorithm are as follows:

1. object-based granularity
2. lock-based STM
3. visible reads
4. lazy update of shared objects
5. lazy acquisition of objects

In face of a conflict, an STM should ensure that the conflict is resolved by canceling the effects of operations performed by the particular transaction. Most of the existing implementations use abort mechanism to resolve conflicts, by aborting the transaction(s) in conflict, and re-starting it. In this process, all the work done by the transaction is lost, and the transaction is re-executed from scratch. The main idea behind $CaPR^+$ Algorithm is the use of partial rollback mechanism for conflict resolution. Partial rollback utilizes some of the work already done by the transaction, and instead of re-executing the transaction from the start, it rolls back to the latest intermediate point that resolves the conflict, and continues execution from there.

Consider a live transaction T_i which has read a value u for t-object x. Suppose a transaction T_j writes a value v to t-object x and commits. When T_i executes the next memory operation (after c_j), T_i is rolled back to the step before the read of x. We denote that T_j has *invalidated* T_i's read of x. T_i then reads x again.

The following example illustrates this idea. Consider the following history:
$H1 : r_1(x,0)r_2(x,0)r_1(y,0)r_1(z,0)w_2(y,10)c_2w_1(x,5)$.
In this history, when T_1 performs any other memory operation such as a read after C_2, it will then be rolled back to the step $r_1(y)$ causing it to read y again. Rolling back to an earlier point, say $r_1(x,0)$ also resolves the conflict, but would redundantly read the value of x as 0. On the other hand, if it rolls back to a later point, this

action does not resolve the conflict, as the value of y read by $T_1(0)$ is now stale, and may cause problems later.

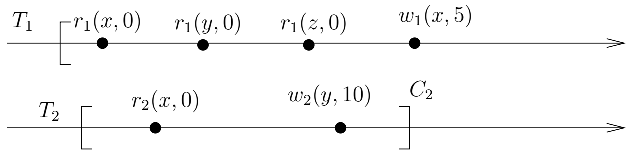


Figure 2-1: Pictorial representation of History $H1$

An important point to note is that partial rollback necessitates creation of checkpoints at intermediate points, to which the transactions could rollback to, in case of a conflict. Each transaction is executed speculatively, since all the updates on a shared object occur to its local copy stored by the STM. The actual updates to the shared objects are effected only during the commit operation. All these data - checkpoints, copies of local and shared objects etc. are stored by $CaPR^+$ in data structures maintained by it. The data structures used by the $CaPR^+$ Algorithm are discussed in the following section.

2.2.1 Data Structures used in $CaPR^+$ Algorithm

The various data structures used in the $CaPR^+$ Algorithm are categorized into local workspace and global workspace, depending on whether the data structure is visible to the local transaction or every transaction. The data structures in local workspace start with letter l - referring to local, and g for those in global workspace. The data structures used in the local workspace are as follows:

1. Local Data Block (*l_LDB*) - whenever a transaction reads a local data object for the first time, an entry is added to its *l_LDB*. Each entry in the local data block consists of the local object and its current value in the transaction (Table 2.2).

2. Shared Object Store (*l_SOS*) - Each entry in the *l_SOS* (shown in Table 2.3) stores the address of the t-object, its value, a read flag, and a write flag. Both the read and write flags have 0 as their initial value. Value 1 in read/write

flag indicates the object has been read/written by the transaction, respectively. Whenever a transaction reads a shared object for the first time, an entry is added to the Shared Object Store of the transaction, and the Read flag of this entry is set to 1. Any update to the shared object by the transaction is stored locally in the value field of *l_SOS*, and the Write flag is set to 1. The Read/Write flag is used by the commit operation to determine the transaction's read/write set, respectively.

3. Checkpoint Log (*l_CPLog*) - Whenever a transaction reads a shared object for the first time, a checkpoint is created that captures the state of the *l_LDB* and *l_SOS* at that point of time. These checkpoints are later used to partially rollback the transaction in case of a conflict. The *l_CPLog* (Table 2.4) keeps a log of all the checkpoints, where each entry stores:

    (a) the shared object whose read initiated the log entry,

    (b) the program location from which the transaction should proceed after a rollback, and

    (c) the current snapshot of the transaction's local data block and the shared object store.

Table 2.2: Local Data Block

Object	Value
100	24
200	10

Table 2.3: Shared object Store

Object	Current Value	Read flag	Write flag
1100	20	1	0
1500	30	1	1

The data structures in the global workspace are:

Table 2.4: Checkpoint Log

Victim Shared Object	Program Location	LDB Snapshot	SOS Snapshot
1100	1	(100, 24)	NULL
1500	2	(100, 24), (200, 10)	(1100, 20)

Table 2.5: Global List of Active Transactions

Transaction ID	Status Flag	Conflict Objects
1	1	1100, 1500
2	0	NULL

1. Global List of Active Transactions (*g_Actrans*) - Each entry in this list (Table 2.5) contains a) a unique transaction identifier, b) a status flag that indicates the status of the transaction, as to whether the transaction is in conflict with any of the committed transactions, and c) a list of all the objects in conflict with the transaction. This list is updated by the committed transactions.

2. Shared Memory (*g_SM*) - Each entry in the shared memory (Table 2.6) stores a) a shared object, b) its value, and c) an active readers list that stores the transaction IDs of all the transactions reading the shared object.The acreaders list is used by a committing transaction to identify the conflicting transactions.

2.2.2 Description of $CaPR^+$ Algorithm

The $CaPR^+$ algorithm is shown in Algorithm 1. As discussed before, with an STM, programmers just need to

- identify atomic blocks within the application,
- demarcate them as transactions, and
- replace all shared memory accesses within the transaction by STM calls for reading/writing the objects. (This could also be performed by an STM compiler, which would reduce the programming complexity but may also result in

Table 2.6: Shared Memory

Shared object	Value	List of active readers
1100	20	1, 2
1500	10	1, 2

degrading performance due to over-instrumentation.)

The various functions defined in the CaPR+ algorithm are: TM_Begin, TM_End, ReadTx, WriteTx, CommitTx, and Partially_Rollback. The description of the various functions defined by the $CaPR^+$ algorithm is given below:

1. ReadTx: When reading object o, the transaction first checks for the presence of o in its *l_LDB*, and if present, its value is returned. If not, *l_SOS* is checked. If the object is present in neither *l_LDB* nor *l_SOS*, the read request is directed to the shared memory. Since o is being read for the first time by the transaction, an entry is created in the *l_SOS* that stores a copy of o, and the entry's read flag is set to 1. A checkpoint entry is also created and stored in the *l_CPLog*. Finally, to make the reader visible, the transaction's id, t, is added to o's readers' list. When the object is found in either *l_LDB* or *l_SOS*, its value is stored in the output structure str that is returned by the function. The transaction then retrieves the value of the object from str.

2. WriteTx: A write operation simply updates locally the value of the local/shared object in the corresponding entry of o in *l_LDB*/*l_SOS*. If the object is found in *l_SOS*, its write flag is set to 1.

3. commitTx: The commit operation is not visible to the programmer, it is invoked implicitly at the end of a transaction by the EndTx call. It first finds the write-set, $t.WS$ of the transaction by looking for objects in *l_SOS* with write flag = 1. To prevent deadlock, objects in $t.WS$ are sorted and locks are obtained in this order. This prevents deadlock by ensuring that the circular wait condition for a deadlock to occur never holds true. Next, it finds the active readers of all objects in $t.WS$ by referring to *g_SM*, sorts them and obtains locks on them.

On successfully acquiring all the locks, the transaction updates all the values of shared objects from *l_SOS* to Shared Memory (*g_SM*). Now, in order to maintain consistency, all the conflicting transactions need to be rolled back. Towards this, all the transactions which have read any of the objects written by the transaction, t, are made to roll back by setting their status flags to RED, and their conflict objects' list is pointed to $t.WS$. This information is later used by the conflicting transactions to identify the appropriate checkpoint to which it needs to rollback. Finally, for every object o read by t, t is removed from o's active readers list, and all the locks acquired are released.

4. Partially_Rollback: Whenever a transaction finds its status flag to be RED, it rolls back by identifying the appropriate checkpoint and applying it. In order to identify the safest checkpoint, it looks at the conflict objects list in *g_Actrans*, and finds the object, o in the conflict objects list that has been read earliest by t. With this o, transaction t looks into the *cplog* entry of o for the checkpoint associated with object o. This is the safest checkpoint, that restores the consistency of the transaction. The corresponding snapshot is applied to its *l_LDB* and *l_SOS*. Finally, its status flag is set to GREEN, and returns the new program location from which it continues the execution. To apply a checkpoint, the state of *l_LDB* and *l_SOS* captured in the snapshot is restored in the *l_LDB* and *l_SOS* data structures and the value of the transaction's program counter is replaced with the corresponding value in the selected checkpoint's Program Location. Finally, all the subsequent entries in the checkpoint log are deleted.

We have implemented the $CaPR^+$ algorithm as a library in C^+. Our implementation uses the concurrent containers provided by Intel's Threading Building Blocks (TBB) library [74] to allow multiple threads to concurrently access and update the shared containers, safely. In particular, these concurrent containers are used to implement the data structures in the global workspace, which are shared among all the transactions. We discuss the performance of $CaPR^+$ in detail in Section 3.2.

```
1: procedure ReadTx($t, o, pc, str$)
```

```
    if o is in t's local data block then
        str.val ← o.val from l_LDB;
        return $l \leftarrow 1(Success)$;
    else if o is in t's shared object store then
        str.val ← o.val from l_SOS;
        return $l \leftarrow 1(Success)$;
    else if o is in shared memory then
        obtain locks on o, & the entry for t in g_Actrans;
        if t.status_flag = RED then
            Unlock locks on t and o;
            $PL = partially\_Rollback(t)$;
            update str.PL = PL;
            return $l \leftarrow 0(Rollback)$;
        create_checkpoint(o, pc);
        str.val ← o.val from Shared Memory;
        add t to o's readers' list;
        add o into l_SOSand set its read flag to 1;
        release locks on o and t;
        return $l \leftarrow 1(Success)$;
    else                                            ▷ o not in shared memory
        return $l \leftarrow 2(Error)$;
procedure WriteTx($o, t$)
    if o is a local object then
        update o in local data block;
    else if o is a shared object then
        if o is in l_SOS then
            update o in l_SOSand set its write flag to 1;
        else
            insert o in l_SOSand set its write flag to 1;
```

```
31: procedure COMMITTX($t$)
32:     Assign t's write-set, $t.WS = \{o|o$ is in l_SOSand o's write flag $= 1;\}$
33:     Sort t.WS, obtain locks on all objects in t.WS;
34:     Initialize $A = \{t\}$;
35:     for each object o in the t.WS
36:         $A = A \cup$ active readers of o;
37:     Sort 'A', obtain locks on all transactions in A;
38:     if t.status_flag = RED then
39:         release all locks;
40:         $PL = partially\_Rollback(t)$;
41:         update str.PL = PL;
42:         return $l \leftarrow 0(Rollback)$;
43:     for each object, wo in t.WS
44:         update wo.value in SM $\leftarrow$ local copy of wo;
45:         for each transaction rt in wo's active readers' list,
46:             rt's conflict objects' list = rt's conflict objects' list $\cup$ t.WS;
47:             set rt's status flag to RED;
48:     delete t from actrans;
49:     for each object, ro in t's readers-list
50:         delete t from ro's active readers list;
51:     release all locks;
52:     return 0;
53: procedure PARTIALLY_ROLLBACK($t$)
54:     find the conflicting write-set from g_Actrans
55:     identify safest checkpoint (earliest conflicting object read by t);
56:     find the l_CPLog entry of the victim object, apply the selected checkpoint;
57:     delete t from active reader's list of all objects rolled back;
58:     reset status flag to GREEN;
59:     nullify conflict pointer of t's g_Actrans entry;
```

```
60:     return PL (the new program location);
```

Implementation of Partial Rollback

With abort, a transaction discards all the work already done by it and restarts execution from beginning. However with partial rollback, a transaction must first restore the execution context and then roll back to the latest point within the transaction that resolves the conflict. As discussed in the earlier section, restoring of execution context is accomplished using the snapshots stored in *l_CPLog*. Rollback to the appropriate checkpoint entails transfer of control. This can be achieved either using setjmp/longjmp or goto. Although, in general, the use of goto is discouraged, its usage greatly simplifies the implementation. Implementing rollback using loops is also possible, but it makes the code quite complicated due to multiple loop-control/return statements as exit points. Moreover, with large applications, the code can become clumsy and unmanageable. However, the main drawback of goto is that it allows transfer of control only within a function. Thus, if a function is called from within a transaction, goto cannot be used. The only way to overcome this problem is to inline all the functions. Although, this is possible for applications with small transactions, the code becomes clumsy for other applications, and is also very hard to debug.

Longjmp is a better alternative that implements non-local goto. It allows control to be transferred from one function to another. The initial version of our implementation was realized using longjmp/setjmp. However, once you return from a function, you cannot jump back to a point within the function, without corrupting the stack. This may lead to inconsistent results, and may even crash the application.

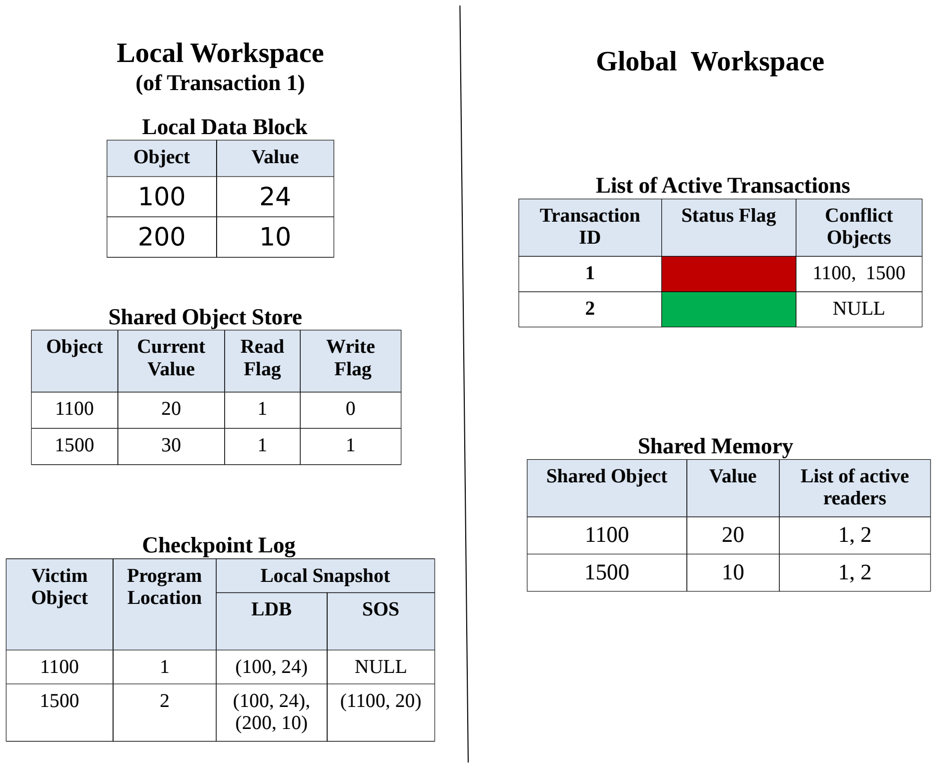
This convinced us to implement rollback using a combination of both setjmp/-longjmp and goto. With this approach, a transaction saves the execution context at its beginning, and contines execution. Since goto is used, we use labels to identify checkpoints that occur at read operations. Labels are also assigned to function calls. Now, there may be arbitrary levels of nesting of functions, which forces us to use a data structure that stores the program counter, pc. pc essentially captures the labels associated with reads and function calls. The state of pc is recorded by transactions in *l_CPLog* while capturing the snapshots. In case of a conflict, the transaction uses longjmp to jump to the start of the transaction, and then goto is used to jump to the appropriate label. Rollback is accomplished successfully when the pc stored in the snapshot of the appropriate *l_CPLog* entry matches the current pc, at which point, the rollback flag is set to 0, and execution continues as usual. Local jump is realized using a combination of labels as values, a static array that serves as a jump table, and the computed goto statement. We show a simplified code snippet below of a sample transaction that illustrates the implementation.

```
1: TM_BEGIN();
2:       Static void *array={&&c0, &&c1, ..., &&c4};
3:       int val = setjmp(env);
4:       if(rollback == 1)
5:              goto *array[pr];
6:       c0: func1();
             .
             .
             .
15:      c4: start = (int)TM_SHARED_READ(global_i);
16:      if(rollback == 1)
17:             longjmp(env, 1);
18:      TM_SHARED_WRITE(global_i, (global_i + 1));
19: TM_END();
```

The usage of goto and longjmp/setjmp to implement rollback necessitates supplementary code. In this listing, the additional code inserted is in lines 2-5 and 16-17. On detecting a conflict at line 16, the transaction jumps to line 3 using longjump, and then jumps locally to the appropriate checkpoint captured in *pr*, using goto. With minor additions, this works well even with codes with arbitrary levels of nesting of functions. Please note that even though the listing above looks complex, most of the additional code has been abstracted out using macros in our implementation, with only marginal inputs required from the user. We now give an illustration of the working of $CaPR^+$ for a better understanding.

An illustration of the working of $CaPR^+$

Consider two concurrent transactions T1 and T2, with transaction IDs 1 and 2 respectively. Figure 2-2 shows the state of data structures in the local workspace of T1, and the data structures of the global workspace. Two transactions conflict if their read/write sets overlap, and there is at least one write operation in common.



Local Workspace
(of Transaction 1)

Local Data Block

Object	Value
100	24
200	10

Shared Object Store

Object	Current Value	Read Flag	Write Flag
1100	20	1	0
1500	30	1	1

Checkpoint Log

Victim Object	Program Location	Local Snapshot	
		LDB	SOS
1100	1	(100, 24)	NULL
1500	2	(100, 24), (200, 10)	(1100, 20)

Global Workspace

List of Active Transactions

Transaction ID	Status Flag	Conflict Objects
1	🟥	1100, 1500
2	🟩	NULL

Shared Memory

Shared Object	Value	List of active readers
1100	20	1, 2
1500	10	1, 2

Figure 2-2: Local workspace of T1 and the global workspace.

All the updates made to the local objects are recorded in *l_LDB*, while *l_SOS* stores a copy of all the shared objects read by the transaction. Any update to any of the shared objects is effected on *l_SOS* rather than on *g_SM*. On encountering a readTx operation, the transaction checks for the presence of that object in *l_LDB*, and then in *l_SOS*. If the object is present in any of them, its value is immediately returned. Otherwise, the transaction creates an entry for this object in *l_CPLog*, and creates a checkpoint i.e., it stores the current state of *l_SOS* and *l_LDB* into the local snapshot field of *l_CPLog*, and the current program location (the transaction rolls back to this location in case of a conflict due to this object). The transaction then looks into *g_SM*, makes a copy of the object in *l_SOS*, and sets its Read Flag to 1. Any further calls of writeTx to this object modifies its copy in *l_SOS*.

Figure 2-2 shows a situation where transaction T1 and T2 are in conflict, and T2, in its commit phase, has changed T1's status flag to RED and stored T2's write set in the conflict objects' list of T1's *g_Actrans* entry. Now, as soon as T1 performs the next memory operation (readTx/commitTx), it finds its status flag to be RED, and rolls back. It first looks at the conflict objects' list to identify the victim object (the object which was read earliest by T1) - in this case object with address 1100. It then looks for the *l_CPLog* entry of the victim object and applies the checkpoint by restoring the state of *l_LDB* and *l_SOS* to that specified in the local snapshot. T1 then deletes the irrelevant entries from *l_LDB*, *l_SOS* and *l_CPLog*, and changes its status flag to GREEN, completing the rollback process. T1 then rolls back to the program location specified in the *l_CPLog* entry and proceeds with execution from this location.

We now give a detailed account of the various correctness criteria for STMs and then give a proof of correctness of $CaPR^+$.

2.2.3 Correctness of Transactional Memories

From a programmers point of view, transactions appear like critical sections that are protected by global locks. It appears that all transactions are executed sequentially, and the aborted transactions are rolled back completely. However, in practice, transactions do not run sequentially. TMs are supposed to make use of the parallelism provided by the underlying hardware, and should not limit the parallelism of transactions executed by different processes or threads. A history contains sequences of interleaved events from multiple concurrent transactions. Although executing the transactions in parallel may help improve the performance of the application by efficiently utilizing the underlying hardware, the correctness may be compromised with. To this end, it becomes important to specify correctness guarantees expected of the different TM implementations.

Correctness criteria from databases provide some intuition for the semantics of TM systems, but they do not consider all of the complexities. First, they specify how committed transactions behave, but they do not define what happens while a transaction runs. Second, criteria such as serializability assume that the database mediates on all access to data, and so they do not consider cases where data is sometimes accessed via transactions and sometimes accessed directly.

Some of the correctness criteria defined for databases are:

1. linearizability [46]

   each of these operations appears to execute atomically at some point between when it is invoked and when it completes.

2. Serializability [47]

   It states that the result of executing concurrent transactions on a database must be identical to a result in which these transactions executed serially.

3. Strict Serializability [48]

   It requires that if transaction T1 completes before transaction T2 starts, then T1 must occur before T2 in the equivalent serial execution.

4. Snapshot Isolation [49]

   It allows the reads that a transaction performs to be serialized before the transactions writes. The reads must collectively see a valid snapshot of memory, but SI allows concurrent transactions to see that same snapshot and then to commit separate sets of updates that conflict with the snapshot but not with one another.

In [50], a correctness criterion for TM, called opacity has been proposed. While strict serializability preserves the order of conflicting operations between transactions, and the order of non-overlapping transactions, Opacity ensures, in addition, that aborting(and live) transactions do not see an inconsistent state of the memory, which can be disastrous in STMs.

It is a safety property that captures the following requirements:

1. all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime.

2. no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and

3. every transaction always observes a consistent state of the system.

The reason why it requires aborted and live transactions to access only consistent states is that a transaction that accesses an inconsistent state can cause various problems, even if it is later aborted. This can be illustrated by the following example-Consider two shared objects x and y. A programmer may require that y is always equal to x^2, and $x \geq 2$. Clearly, the programmer will then take care that every transaction, when executed as a whole, preserves the assumed invariants. Assume the initial value of x and y is 4 and 16, respectively. Let T1 be a transaction that performs the following operations:

```
$x.write(2);$
$y.write(4);$
$tryC \rightarrow C1;$
```

Consider another transaction T2 that is executed by some process pi concurrently with T1. Then, if T2 reads the old value of x (4) and the new value of y (also 4), the following problems may occur, even if T2 is to be aborted later.

If T2 tries to compute the value of $1/(y - x)$, then a divide by zero exception will be thrown, which can crash the process executing the transaction or even the entire application.

Also, if pi enters the following loop:

```
$vx \leftarrow x.read;$
$vy \leftarrow y.read;$
$while\ vx = vy\ do$
$array[vx] \leftarrow 0;$
```

$vx \leftarrow vx + 1;$

and pi reads $vx = vy = 4$, then unexpected memory locations could be overwritten, apart from the fact that the loop would need to span the entire value domain.

Formally, opacity can be defined as:

Definition 1 *A history H is opaque if there exists a sequential history S equivalent to some history in set Complete(H), such that*

1. *S preserves the real-time order of H, and*
2. *every transaction Ti ϵ S is legal in S.*

Virtual World Consistency [119] is another correctness criteria that is similar to opacity for committed transactions, but weaker than opacity for aborted transactions. It states that

1. no transaction (committed or aborted) reads values from an inconsistent global state,
2. the consistent global states read by the committed transactions are mutually consistent (in the sense that they can be totally ordered)
3. while the global state read by each aborted transaction is consistent from its individual point of view, the global states read by any two aborted transactions are not required to be mutually consistent.

In [116], 2 correctness criteria have been introduced - TMS1 and TMS2. TMS1 requires the behavior observed by committed transactions to be justified by a single execution, while allowing active and aborted ones to be justified by different executions. TMS2 is less permissive than TMS1. In [115], Lesani et. al. proved that opacity is weaker than TMS2 but stronger than TMS1.

We now establish the correctness of $CaPR^+$ algorithm by giving the proof of opacity for $CaPR^+$. We used conflict opacity, a stronger criteria than opacity, to arrive at the proof. In particular, we first show the graph characterization of conflict opacity, i.e., a history H is co-opaque if the corresponding conflict graph is acyclic. We then show that for any history generated by $CaPR^+$ algorithm, the corresponding conflict graph is acyclic. This shows that every history generated by $CaPR^+$ algorithm is co-opaque, and hence opaque.

We now give the formal definition of conflict opacity, show the graph characterization of conflict opacity, and then give the proof of opacity.

Conflict Opacity

In this section we describe *Conflict Opacity* (CO), a subclass of opacity using conflict order (defined in Section 2.1.2). This subclass is similar to conflict serializability for databases, whose membership can be tested in polynomial time (in fact it is closer to order conflict serializability) [72, Chap 3].

Definition 2 *A history H is said to be* conflict opaque *or* co-opaque *if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$ and $\prec_H^{CO}$.*

From this definition, we can see that co-opaque is a subset of opacity.

Graph characterization of co-opacity

Given a history H, we construct a *conflict graph*, $CG(H) = (V, E)$ as follows: (1) $V = txns(H)$, the set of transactions in H (2) an edge (T_i, T_j) is added to E whenever $T_i \prec_H^{RT} T_j$ or $T_i \prec_H^{CO} T_j$, i.e., whenever T_i precedes T_j in the real-time or conflict order.

Note, since $txns(H) = txns(\overline{H})$ and $(\prec_H^{RT} \cup \prec_H^{CO}) = (\prec_{\overline{H}}^{RT} \cup \prec_{\overline{H}}^{CO})$, we have $CG(H) = CG(\overline{H})$. In the following lemmas, we show that the graph characterization indeed helps us verify the membership in co-opacity.

Lemma 1 *Consider two histories $H1$ and $H2$ such that $H1$ is equivalent to $H2$ and $H1$ respects the conflict order of $H2$, i.e., $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. Then, $\prec_{H1}^{CO} = \prec_{H2}^{CO}$.*

Proof. Here, we have that $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. In order to prove $\prec_{H1}^{CO} = \prec_{H2}^{CO}$, we have to show that $\prec_{H2}^{CO} \subseteq \prec_{H1}^{CO}$. We prove this using contradiction. Consider two events p, q belonging to transaction $T1, T2$ respectively in $H2$ such that $(p, q) \in \prec_{H2}^{CO}$ but $(p, q) \notin \prec_{H1}^{CO}$. Since the events of $H2$ and $H1$ are same, these events are also in $H1$. This implies that the events p, q are also related by CO in $H1$. Thus, we have that either $(p, q) \in \prec_{H1}^{CO}$ or $(q, p) \in \prec_{H1}^{CO}$. But from our assumption, we get that the former is not possible. Hence, we get that $(q, p) \in \prec_{H1}^{CO} \Rightarrow (q, p) \in \prec_{H2}^{CO}$. But we already have that $(p, q) \in \prec_{H2}^{CO}$. This is a contradiction.

Lemma 2 *Let $H1$ and $H2$ be equivalent histories such that $\prec_{H1}^{CO} = \prec_{H2}^{CO}$. Then $H1$ is legal iff $H2$ is legal.*

Proof. It is enough to prove the 'if' case, and the 'only if' case will follow from symmetry of the argument. Suppose that $H1$ is legal. By contradiction, assume that $H2$ is not legal, i.e., there is a read operation $r_j(x, v)$ (of transaction T_j) in $H2$ with lastWrite as c_k (of transaction T_k) and T_k writes $u \neq v$ to x, i.e., $w_k(x, u) \in evts(T_k)$. Let $r_j(x, v)$'s lastWrite in $H1$ be c_i of T_i. Since $H1$ is legal, T_i writes v to x, i.e., $w_i(x, v) \in evts(T_i)$.

Since $evts(H1) = evts(H2)$, we get that c_i is also in $H2$, and c_k is also in $H1$. As $\prec_{H1}^{CO} = \prec_{H2}^{CO}$, we get $c_i <_{H2} r_j(x, v)$ and $c_k <_{H1} r_j(x, v)$.

Since c_i is the lastWrite of $r_j(x, v)$ in $H1$ we derive that $c_k <_{H1} c_i$ and, thus, $c_k <_{H2} c_i <_{H2} r_j(x, v)$. But this contradicts the assumption that c_k is the lastWrite of $r_j(x, v)$ in $H2$. Hence, $H2$ is legal. From the above lemma we get the following interesting corollary.

Corollary 3 *Every co-opaque history H is legal as well.*

Based on the conflict graph construction, we have the following graph characterization for co-opaque.

Theorem 4 *A legal history H is co-opaque iff $CG(H)$ is acyclic.*

Proof.

(Only if) If H is co-opaque and legal, then $CG(H)$ is acyclic: Since H is co-opaque, there exists a legal t-sequential history S equivalent to $\overline{H}$ and S respects $\prec_H^{RT}$ and $\prec_H^{CO}$. Thus from the conflict graph construction we have that $CG(\overline{H})(= CG(H))$ is a sub graph of $CG(S)$. Since S is sequential, it can be inferred that $CG(S)$ is acyclic. Any subgraph of an acyclic graph is also acyclic. Hence $CG(H)$ is also acyclic.

(if) If H is legal and $CG(H)$ is acyclic then H is co-opaque: Suppose that $CG(H) = CG(\overline{H})$ is acyclic. Thus we can perform a topological sort on the vertices of the graph and obtain a sequential order. Using this order, we can obtain a sequential schedule S that is equivalent to $\overline{H}$. Moreover, by construction, S respects $\prec_H^{RT} = \prec_{\overline{H}}^{RT}$ and $\prec_H^{CO} = \prec_{\overline{H}}^{CO}$.

Since every two events related by the conflict relation (w-w, r-w, or w-r)in S are also related by $\prec_{\overline{H}}^{CO}$, we obtain $\prec_S^{CO} = \prec_{\overline{H}}^{CO}$. Since H is legal, $\overline{H}$ is also legal. Combining this with Lemma 2, we get that S is also legal. This satisfies all the conditions necessary for H to be co-opaque.

Proof of Opacity for $CaPR^+$ Algorithm

In this section, we will describe some of the properties of $CaPR^+$ algorithm and then prove that it satisfies opacity. In our implementation, only the read and tryC operations access the memory. Hence, we call these memory operations. The main idea behind our algorithm is that when a transaction's read is invalidated, it does not abort but rather gets rolled back to an intermediate point. In the worst case, it could get rolled back to the first step of the transaction which is equivalent to the transaction being aborted and restarted. Thus with this algorithm, a history will consist only of incomplete (live) and committed transactions.

To precisely capture happenings of the algorithm and to make it consistent with the model we have discussed so far, we modify the representation of the transactions that are rolled back. Consider a transaction T_i which has read x. Suppose another transaction T_j writes to x and then commits. Thus, when T_i performs its next memory operation, say m_i (which could either be a read or commit operation), it will be rolled back. We capture this rollback operation in the history as two transactions: $T_{i.1}$ and $T_{i.2}$.

Here, $T_{i.1}$ represents all the successful operations of transaction T_i until it executed the memory operation m_i which caused it to roll back (but not including m_i). Transaction $T_{i.1}$ is then terminated by an abort operation $a_{i.1}$. Then, after transaction T_j has committed transaction $T_{i.2}$ begins. Unlike $T_{i.1}$ it is incomplete. It also consists of all same operations of $T_{i.1}$ until the read on x. $T_{i.2}$ reads the latest value of the t-object x again since it has been invalidated by T_j. It then executes future steps which could depend on the read of x. With this modification, the history consists of committed, incomplete as well as aborted transactions (as discussed in the model).

In reality, transaction T_i could be rolled back multiple times, say n. Then the history H would contain events from transactions $T_{i.1}, T_{i.2}, T_{i.3} .... T_{i.n}$. But it must be noted that all the invocations of T_i are related by real-time order. Thus, we have that $T_{i.1} \prec_H^{RT} T_{i.2} \prec_H^{RT} T_{i.3} .... \prec_H^{RT} T_{i.n}$

With this change in the model, the history $H1$ is represented by $H2$ as follows,
$H2 : r_{1.1}(x,0)r_{2.1}(x,0)r_{1.1}(y,0)r_{1.1}(z,0)w_{2.1}(y,10)c_{2.1}w_1(x,5)a_{1.1}r_{1.2}(x,0)r_{1.2}(y,10)$.

For simplicity, from now on in histories we will denote a transaction with a Greek letter subscript such as α, β, γ, etc., regardless of whether it is invoked for the first time or has been rolled back. Thus in our representation, transactions $T_{i.1}$ and $T_{i.2}$ could be denoted as T_α and T_β respectively.

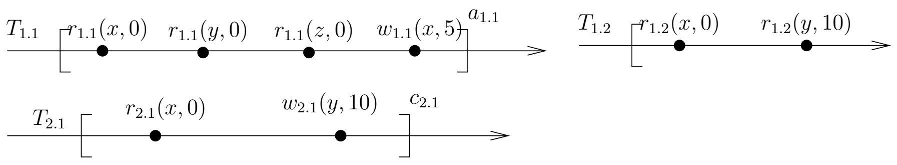


Figure 2-3: Pictorial representation of the modified History $H2$

We will now prove the correctness of this algorithm. We start by describing a property that captures the basic idea behind the working of the algorithm.

Property 5 *Consider a transaction T_i that reads t-object x. Suppose another transaction T_j writes to x and then commits. In this case, the next memory operation* *(read or tryC) executed by T_i after c_j returns abort (since the read of x by T_i has been invalidated).*

For a transaction T_i we define the notion of its *successful final memory operation(sfm)*. As the name suggests, it is the last successfully executed memory operation of T_i. If T_i is committed, then $sfm_i = c_i$. If T_i is aborted, then sfm_i is the last memory operation, in this case a read operation, that returned *ok* before being aborted.

For proving correctness, we use the graph characterization of co-opacity described in Section 2.2.3.

Consider a history H_{capr} generated by the $CaPR^+$ algorithm. Let $CG(H_{capr})$ be the conflict graph of H_{capr}. We show that this graph denoted, g_{capr}, is acyclic.

Lemma 6 *Consider a path p in g_{capr} abstracted as: $T_{\alpha 1} \rightarrow T_{\alpha 2} \rightarrow .... \rightarrow T_{\alpha k}$. Then, $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2} <_{H_{capr}} .... <_{H_{capr}} sfm_{\alpha k}$.*

Proof. We prove this using induction on k.

Base Case, $k = 2$. In this case the path consists of only one edge between transactions $T_{\alpha 1}$ and $T_{\alpha 2}$. Let us analyse the various types of edges possible:

- *real-time edge:* This edge represents real-time. In this case $T_{\alpha 1} \prec^{RT}_{H_{capr}} T_{\alpha 2}$. Hence, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.

- *w-w edge:* This edge represents w-w order conflict. In this case both transactions $T_{\alpha 1}$ and $T_{\alpha 2}$ are committed and $sfm_{\alpha 1} = c_{\alpha 1}$ and $sfm_{\alpha 2} = c_{\alpha 2}$. Thus, from the definition of this conflict, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.

- *w-r edge:* This edge represents w-r order conflict. In this case, $c_{\alpha 1} <_{H_{capr}} r_{\alpha 2}(x, v)$ $(v \neq A)$. For transaction $T_{\alpha 1}$, $sfm_{\alpha 1} = c_{\alpha 1}$. For transaction $T_{\alpha 2}$, either $r_{\alpha 2} <_{H_{capr}} sfm_{\alpha 2}$ or $r_{\alpha 2} = sfm_{\alpha 2}$. Thus in either case, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.

- *r-w edge:* This edge represents r-w order conflict. In this case, $r_{\alpha 1}(x, v) <_{H_{capr}} c_{\alpha 2}$ (where $v \neq A$). Thus $sfm_{\alpha 2} = c_{\alpha 2}$. Here, we again have two cases: (a) $T_{\alpha 1}$

terminates before $T_{\alpha 2}$. In this case, it is clear that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$. (b) $T_{\alpha 1}$ terminates after $T_{\alpha 2}$ commits. The working of the algorithm is such that, as observed in Property 5, the next memory operation executed by $T_{\alpha 1}$ after the commit operation $c_{\alpha 2}$ returns abort. From this, we get that the last successful memory operation executed by $T_{\alpha 1}$ must have executed before $c_{\alpha 2}$. Hence, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.

Thus in all the cases, the base case holds.

Induction Case, $k = n > 2$. In this case the path consists of series of edges starting from transactions $T_{\alpha 1}$ and ending at $T_{\alpha n}$. From our induction hypothesis, we know that it is true for $k = n - 1$. Thus, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha(n-1)}$. Now consider the transactions $T_{\alpha(n-1)}, T_{\alpha n}$ which have an edge between them. Using the arguments similar to the base case, we can prove that $sfm_{\alpha(n-1)} <_{H_{capr}} sfm_{\alpha n}$. Thus, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha n}$.

In all the cases, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha n}$. Hence, proved.

Using Lemma 6, we show that g_{capr} is acyclic.

Lemma 7 *Graph, g_{capr} is acyclic.*

Proof. We prove this by contradiction. Suppose that g_{capr} is cyclic. Then there is a cycle going from $T_{\alpha 1} \rightarrow T_{\alpha 2} \rightarrow .... \rightarrow T_{\alpha k} \rightarrow T_{\alpha 1}$.

From Lemma 6, we get that $sfm_{\alpha 1} \rightarrow sfm_{\alpha 2} \rightarrow .... \rightarrow sfm_{\alpha k} \rightarrow sfm_{\alpha 1}$ which implies that $sfm_{\alpha 1} \rightarrow sfm_{\alpha 1}$. Hence, the contradiction.

Theorem 8 *All histories generated by $CaPR^+$ are co-opaque and hence, $CaPR^+$ satisfies the property of opacity.*

Proof. Proof follows from Theorem 5 and Lemma 8.

Thus, it is proved that $CaPR+$ algorithms satisfies Opacity, thereby establishing its correctness.

We have also extensively evaluated $CaPR^+$ algorithm for performance. We discuss it in detail in the next chapter.

Chapter 3

Comparative Performance Evaluation of CaPR+ Algorithm

In this chapter, we discuss the performance evaluation of $CaPR^+$ algorithm on various benchmark applications. In particular, we carry out an extensive comparative performance evaluation of the Abort and Partial Rollback mechanisms for STMs. For purposes of comparison, we have used the state-of-the-art RSTM system and for the Partial Rollback, and we have used our CaPR+ algorithm. We first give an account of the various benchmarks used in evaluating performance of STM implementations.

3.1 Benchmarks for STMs

Although many Hardware Transactional Memory (HTM), Software Transactional Memory(STM) and Hybrid systems have been proposed, their runtime overhead are yet to be reduced to an acceptable mark, so as to enable them to be embraced by the industry and academia. To this end, the measurement of the performance of TM systems plays an important role as incorrect measurement may mislead the researchers who may end up trying to optimize irrelevant aspects of the implementations. Moreover, the main issue in the evaluation of trade-offs in different systems is that the direct comparison is difficult for systems that are based on different programming languages. This motivates the need for a suitable benchmark that enables the empirical evaluation of all the relevant aspects required or expected of a good TM system.

Existing benchmarks may be categorized into micro-benchmarks and individual (or set of) applications. Micro-benchmarks are composed of transactions that execute a few operations on a data structure. These are typically easy to develop, parameterize, and port across systems. They may be useful in cases when a particular aspect of the implementation is to be selectively evaluated. However, they do not allow us to evaluate the whole implementation exhaustively. On the other hand, full applications have transactions that consist of many operations over many data structures, and may include a significant amount of non-transactional code as well.

Red-black trees and hash-tables are examples of micro-benchmarks used for evaluating TM systems. The short and simple transactions of micro-benchmarks are good for testing mechanics of STM itself and comparing low-level details of various implementations. STMBench7 [53] is another candidate benchmark for evaluating STM implementations. Their motivation was to come up with a comprehensive benchmark suite that is able to produce a set of workloads that:

1. corresponds to realistic, complex, object-oriented applications which benefit from multi-threading

2. does not depend on any particular STM or programming language

3. are easy to use and provides results which can be readily interpreted.

However STMBench7 targets only a specific class of applications, i.e., CAD/-CAM. Other benchmarks include SPLASH-2 [54], SPEComp [55], BioParallel [56], MineBench [57] etc, but almost all of them lack in dealing with a wide range of transactional behaviors - contention, length of transactions, and sizes of their read and write sets. This leads us to another benchmark called STAMP [58] (Stanford Transactional Applications for Multi-Processing). It is a comprehensive benchmark suite that includes eight applications spanning different classes of applications, and thirty variants of input parameters and data sets that exercise a wide range of transactional behaviors. We have used STAMP benchmarks extensively for our experiments in this work.

STAMP Benchmarks

Table 3.1: STAMP applications

Application	Domain	Description
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

Stanford Transactional Applications for Multi-Processing (STAMP) is a comprehensive benchmark suite that includes eight applications spanning different classes of applications, and thirty variants of input parameters and data sets that exercise a wide range of transactional behaviors. The main features of benchmarks thrusted upon by the authors are:
1) Breadth - the benchmark must target a variety of algorithms and application domains.
2) Depth - it must cover a wide range of transactional characteristics - transaction lengths, contention and size of read and write sets.
3) The amount of time spent in executing the transactions should be varied.
4) Portability- it must be compatible with a large variety of TM systems.

Table 1 depicts the different applications and their brief description.

3.2 Performance Evaluation of $CaPR^+$ Algorithm

In this section, we evaluate $CaPR^+$ using the kmeans, genome, ssca2, labyrinth and vacation applications of the STAMP benchmark suite [58]. This involves the following steps:

1. To adapt the benchmarks for $CaPR^+$ algorithm, change lib/tm*.
2. Also modify the application code by changing the operations to those defined by the library of $CaPR^+$ algorithm(Read, write etc).
3. Configure and compile the $CaPR^+$ library.
4. Modify STAMP to point to the libstamp directory(that includes the $CaPR^+$ library) :
   (a) In (STAMP)/common/Makefile.stm, change -ltl2 to -lcapr
   (b) In (STAMP)/common/Defines.common.mk, change STM to the relative path to (CaPR+)/libstamp.
5. compile the benchmark application and run it with the appropriate parameters.

To analyze the performance of Partial Rollback mechanism devised for the $CaPR^+$ algorithm, we first modify the original implementation of $CaPR^+$ algorithm to get another version of the same, that aborts the ongoing transaction, and restarts from the beginning of the transaction instead of rolling back to an earlier point(checkpoint). We then use the STAMP benchmark applications and run them with these two versions of $CaPR^+$ algorithm implementation and $RSTM$ [75], and evaluate the performance of the two mechanisms. In particular, we perform a comparative evaluation of $CaPR^+$ under the following variations:

1. We derive a version of $CaPR^+$ implementing pure abort ($CaPR^+(abort)$), and then compare it with $CaPR^+$ (that implements partial rollback) for the STAMP benchmark applications.

2. We comparatively evaluate the performance of $CaPR^{+}$ with $CaPR^{+}(abort)$, with varied transactional delays in the STAMP benchmark applications.

3. We comparatively evaluate the performance of $CaPR^{+}$ with respect to $RSTM$ [75], a state-of-the-art STM implementation, with varied transactional delays in the STAMP benchmark applications.

Experimental Framework

The experiments were performed on the following platform: Intel Xeon E5-2650V4 - comprising 24 cores/48 threads, operating at 2.20 GHz. For concurrent access and update of the shared containers, *Intel TBB 2017 Update 2* was used. The STMs $CaPR^{+}$, $CaPR^{+}(abort)$ and $RSTM$ (Release 7) were compiled using gcc v4.8.4, with *Ubuntu 14.04.5* as the operating system. The results were averaged over a set of 5 test runs, and the standard deviation is displayed as error bars for each experiment. The differences in performance obtained for different iterations are not statistically significant as the variations observed in results are very small as compared to the mean for most of the experiments.Also, the claims we make are for large transactional delays, in which case the standard deviation is negligible as compared to the mean values of the results over different iterations.

3.2.1 Comparative Evaluation of Abort and Partial Rollback mechanisms

In order to analyze the performance of partial rollback mechanism with abort precisely, we derived a light-weight version of the $CaPR^{+}$ implementation that implements abort mechanism ($CaPR^{+}(abort)$), and compare it with the $CaPR^{+}$ implementation. This is realized by eliminating the bookkeeping required by the $CaPR^{+}$ algorithm like checkpointing. The comparison is performed using the kmeans, genome, ssca2, labyrinth and vacation applications of the STAMP benchmarks [58]. We observe the execution times of the benchmark applications with $CaPR^{+}$ and $CaPR^{+}$(*Ab-*

ort) for different number of threads (1, 2, 4, 8, 16, 32, and 64). Each experiment (e.g., execution of the ssca2 benchmark using $CaPR^+$ with 4 threads) is repeated 5 times, and a graph is plotted with the arithmetic mean of the observed execution times. The standard deviations observed for the experiment are displayed as error bars. Note that for most of the observations, the standard deviation is so small that the error bar appears as a small horizontal line.

The results for all experiments in this section have been shown in Figure 3-1. The results show $CaPR^+$(*Abort*) performs much better as compared to $CaPR^+$ in most of the experiments. We discuss results of each of the applications in detail below:

Experiment 1: kmeans - kmeans partitions a data set into k clusters, such that each data object belongs to the cluster with the nearest mean. There are 4 execution configurations for kmeans corresponding to contention (high/low) and input size (small/large), with large input size denoted by '+'. For eg., kmeans-high + denotes the execution configuration with high contention and large input. For kmeans, it is observed that $CaPR^+$(*Abort*) performs better. This can be explained by the small length of transactions in kmeans, which renders rollback less useful. With large inputs, $CaPR^+$ lags significantly behind $CaPR^+$(*Abort*) with one thread, however, the performance gap narrows with increase in threads, with the least performance gap observed with 8 threads. This can be explained by the increase in number of rollbacks/aborts with increase in threads.

Experiment 2: Genome - Genome takes as input a large number of DNA segments, and matches them to reconstruct the original source genome. It uses transactions quite frequently, as it operates continuously on shared data structures and a significant amount of execution time is spent in transactions.

From the results, we observe that for a single thread, the abort mechanism performs well, and then as the number of threads increases, performance of $CaPR^+$ gradually increases. The overall best performance is achieved for 32 threads for $CaPR^+$. Another important point to note is that with more number of threads, there is higher contention, leading to more rollbacks/aborts. This is apparent from the results, and

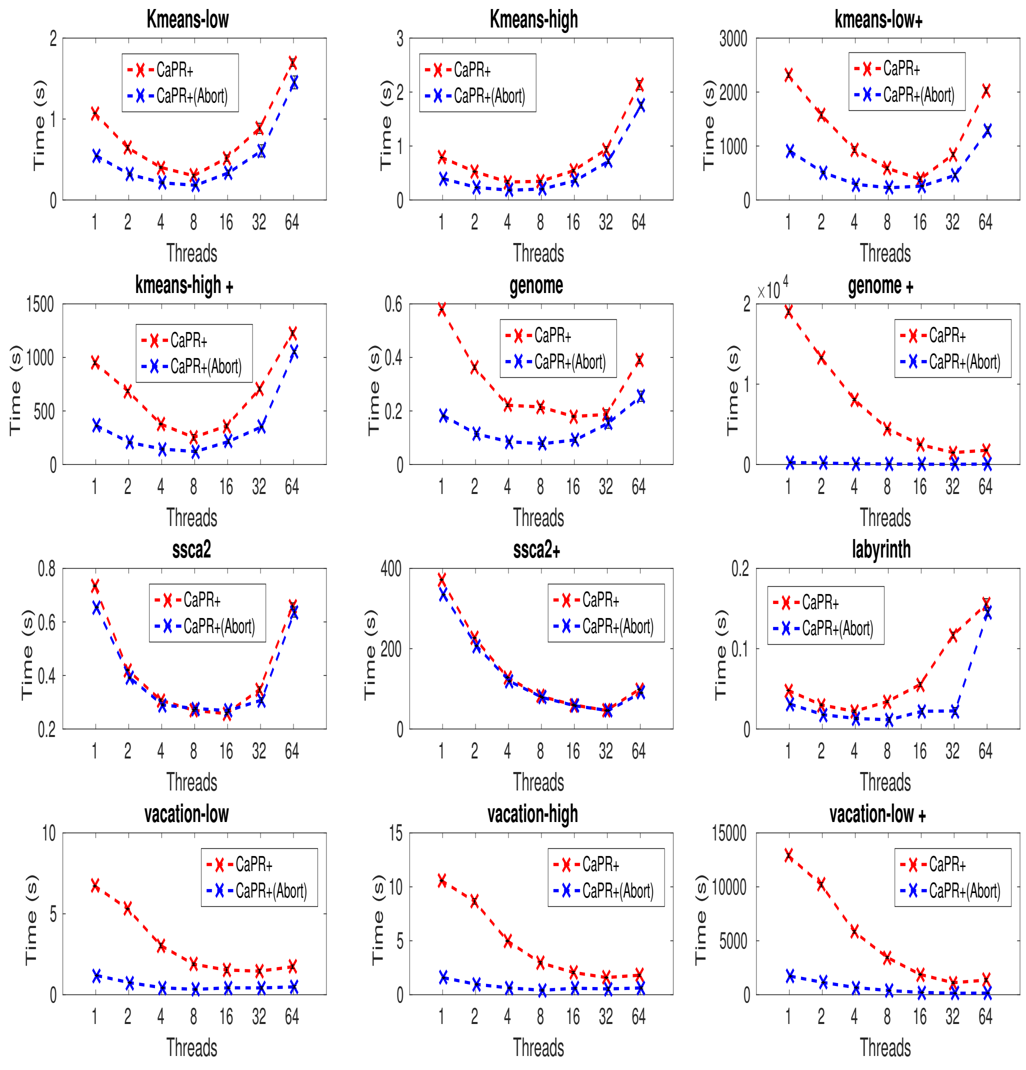


Figure 3-1: Execution time for STAMP benchmark applications with CaPR+ and CaPR+(Abort)

justifies why the performance takes a hit beyond 32 threads. This is also because of over-subscribing as only 48 cores are at disposal.

Experiment 3: SSCA2 - SSCA2 (Scalable Synthetic Compact Applications 2) constructs an efficient graph data structure using adjacency arrays and auxiliary arrays [58].

For ssca2, there is only a marginal difference in performance. This is explained by the fact that the amount of contention is relatively low for ssca2 due to infrequent concurrent updates of the same adjacency list. This is because of the large number of graph nodes resulting in fewer conflicts, and hence fewer aborts/rollbacks.

Experiment 4: Labyrinth- This application is a path finding algorithm based on Lee's algorithm [97] for maze routing. For labyrinth too, abort fares better throughout as compared to rollback. This is because labyrinth has large read and write sets, which adds to the overhead of checkpointing for the rollback mechanism. The best performance is observed for 4/8 threads, and thereafter the performance drops sharply for both rollback and abort. This is because of the high amount of contention due to several transactional accesses to memory.

Experiment 5: Vacation - Vacation implements a travel reservation system, where a number of client threads perform reservation/cancellation/update operations on the database. The results of vacation show a similar trend for every execution, with performance of $CaPR^{+}$ improving gradually with increase in number of threads. Although vacation spends most of its time in transactions, the performance gap remains significantly high even with 16/32 threads. This is due to lesser number of rollbacks witnessed due to low contention among threads.

The performance of $CaPR^{+}$ and $CaPR^{+}(Abort)$ for the above applications show the superiority of the abort mechanism over partial rollback in general. However, the results also suggest that $CaPR^{+}$ performs better than $CaPR^{+}(Abort)$ in applications that witness large number of rollback. It is also observed that for applications with short transactions, $CaPR^+$ trails $CaPR^+(Abort)$, while, for executions with large inputs like genome+, the overhead in checkpointing due to large read sets outweighs the gain achieved due to rollback. This leads us to experiment by modifying these applications to have transactions of varying transactional lengths. This is realized by inserting local transactional delays of varied lengths within transactions. This is because increasing the time spent by the application in transactions results in more conflicts for applications with high contention. Now, for these variants of the STAMP benchmark applications, we compare the performance of $CaPR^+$ with $CaPR^+(abort)$ and RSTM [75] to see how it varies with change in transactional length.

3.2.2 Comparative Performance of $CaPR^+$ with $CaPR^+(abort)$, with varied transactional delays in the STAMP benchmark applications

In this experiment, we compare the performance of $CaPR^+$ with $CaPR^+(abort)$, but with transactional delays introduced in the transactions of STAMP benchmark applications. In particular, we simulate the transactions to resemble varying transactional lengths, by adding arbitrary local transactional code of varying lengths/duration ranging from 0.1 ms to 100ms. Due to infrequent concurrent updates in ssca2 and vacation, introduction of delay in the transactions does not serve any purpose as the number of rollbacks/aborts witnessed in these cases is very small. Hence, we consider only kmeans, genome and labyrinth applications for experiments in this and the following section.

For better understanding of the comparative performance of $CaPR^+$ with respect to $CaPR^+(Abort)$, we plot bargraphs to depict the speedup of $CaPR^+$ with respect to $CaPR^+(Abort)$, for varying local transactional loads. The speedup is given by $t_{CaPR^+(Abort)}/t_{CAPR^+}$, where t_{CaPR^+} is the arithmetic mean of the execution times of $CaPR^+$ for the set of test runs for the particular experiment, and likewise for $t_{CaPR^+(Abort)}$. Thus, a speedup value less than 1 indicates $t_{CaPR^+(Abort)} < t_{CaPR^+}$,

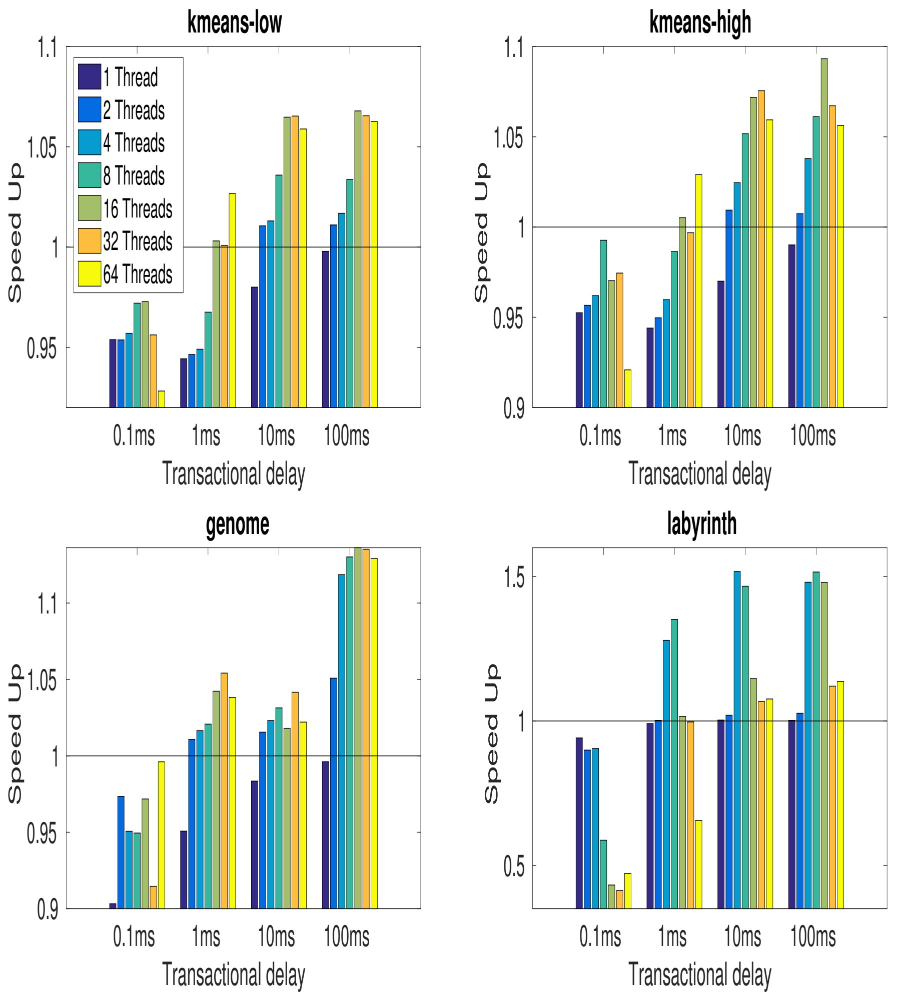


Figure 3-2: Speedup of $CaPR^+$ Algorithm w.r.to $CaPR^+(Abort)$

implying better performance of $CaPR^+(Abort)$, while a speedup greater than one indicates better performance of $CaPR^+$. The results are shown in Figure 3-2.

The results for kmeans show that $CaPR^+(Abort)$ performs much better than $CaPR^+$ for small delays(0.1ms and 1ms). However, for larger delays (10ms and 100ms), $CaPR^+$ performs better than $CaPR^+(Abort)$, with up to 7% and 9% performance gain obtained for kmeans-low and kmeans-high, respectively (Figure 3-2). With genome, the speed-up doesn't gain much with increase in delay to 1ms and 10ms, but with 100ms, the speed-up varies sharply and a performance gain of 13.5% is achieved for $CaPR^+$. For labyrinth, the gain is quite significant, with upto 51% performance gain observed with 8 threads.

An important point to note is that with increase in transactional delay, the performance of $CaPR^+$ gets better as compared to $CaPR^+(abort)$, although the speed-up achieved is not of the same scale as with RSTM (discussed in Section 3.2.3). Another observation is that for every experiment, the speed-up increases with increase in the number of threads, except for with 64 threads. This is because the underlying system consists of only 48 cores.

3.2.3 Comparative performance of $CaPR^+$ with RSTM, with varied transactional delays in the STAMP benchmark applications

In this experiment, we make a comparison of the performance of $CaPR^+$ and RSTM [75], with varying transactional delays in the kmeans, genome and labyrinth applications. As before, we plot bargraphs to depict the speedup of $CaPR^+$ with respect to RSTM, for the kmeans, genome and labyrinth applications of the STAMP benchmark suite, with varying local transactional loads. From the results obtained, we draw the following conclusions:

The execution time of both RSTM and $CaPR^+$ increases uniformly, as the transactional delay increases (by a factor of 10). In the case of RSTM, the execution time increases proportionately, by the factor of 10 with increase in delay. $CaPR^+$ is

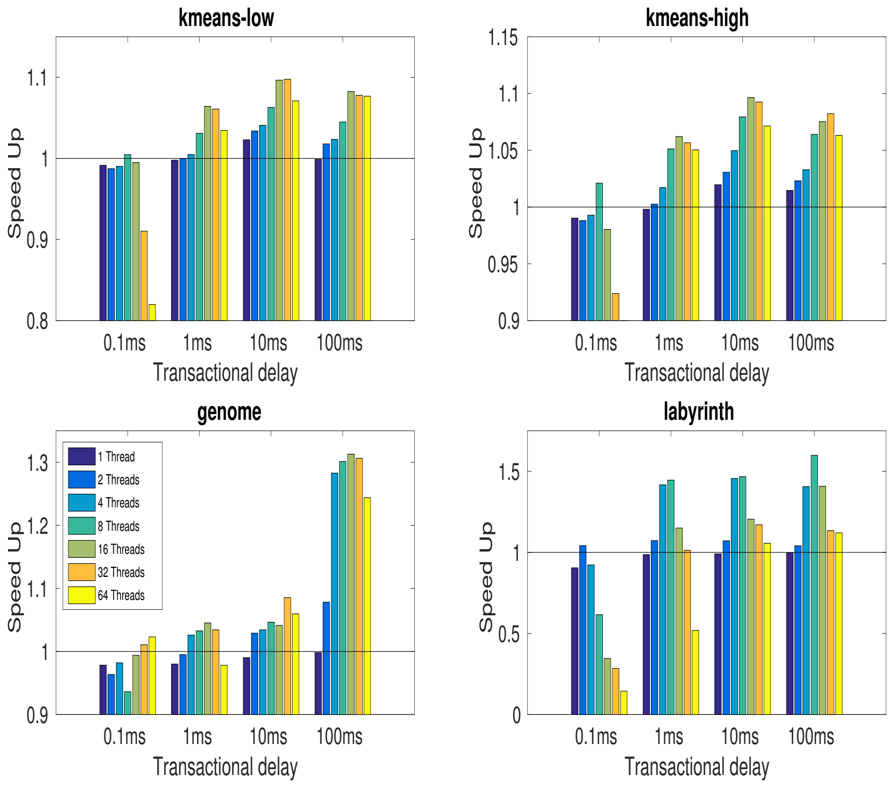


Figure 3-3: Speedup of $CaPR^+$ Algorithm w.r.to RSTM

designed to partially roll back so that on encountering a conflict, the work performed by a transaction is not completely wasted. As a result, for $CaPR^+$, the increase in execution time is not proportional to the increase in the delay. The 10-fold increase in execution time is not witnessed here.

With the introduction of delay in transactions, kmeans now spends a significant amount of its time in transactions. The results for kmeans show that $CaPR^+$ performs better than RSTM for delays greater than 0.1ms in all cases. For both kmeans-low and kmeans high, the maximum performance gain witnessed is 8%.

In the case of genome, as observed in the Section, the performance gain achieved by $CaPR^+$ is visible only for large delays. An important point to note is that although RSTM performs better for small delays, $CaPR^+$ catches up with RSTM with increase in delay, and in fact, fares better than RSTM for 10ms delay with 32 threads. With 100ms delay, the speed-up achieved is an impressive 1.31x. With labyrinth, the speed-up achieved is quite significant (1.6x). The reason for the high speed-up can be attributed to the increased number of aborts witnessed in RSTM.

3.3 Integrated Partial Rollback-Abort Framework

From the results in Section 3.2, we observe that rollback mechanism is expected to perform well with applications in which transaction lengths are relatively longer and the contention among threads is also high. With smaller transactions, the overhead incurred in book-keeping exceeds the performance improvement achieved using rollback. In these circumstances, it makes sense for us to run the transactions using the abort mechanism. Towards this, we propose an integrated abort-partial rollback framework, that provides support for the user to tag some transactions (typically with long transaction length), to run with the partial rollback mechanism, and others to run with the abort mechanism, which entails minimum bookkeeping. This allows us to extract maximum benefit by exploiting both mechanisms simultaneously.

To demonstrate the utility of our idea, we realize the hybrid framework by adapting $CaPR^+$'s interface to also allow support for abort mechanism, such that it allows users to statically tag transactions of an applications to run with either of abort or rollback. An interesting point to note is that this implementation may result in multiple transactions with different mechanisms(abort/rollback) executing concurrently, without any issue.

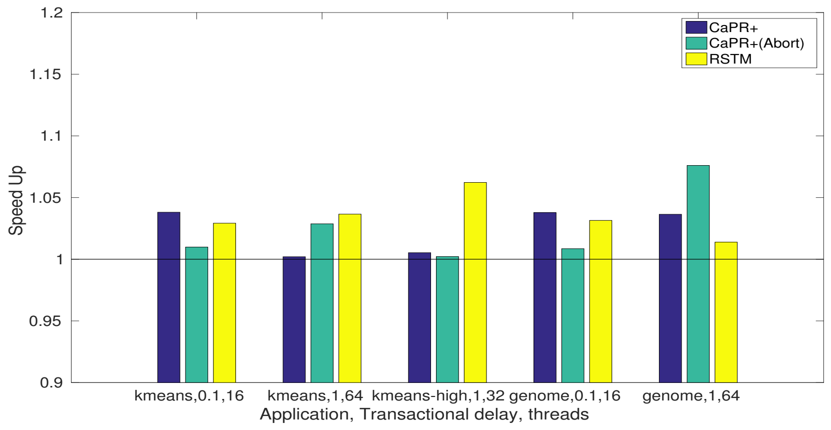


Figure 3-4: Speedup of the hybrid implementation w.r.to $CaPR^{+}$, $CaPR^{+}(Abort)$, and $RSTM$

We plot graphs to depict the speedup of hybrid implementation w.r.to $CaPR^{+}$, $CaPR^{+}(Abort)$, and $RSTM$, for the kmeans, genome and labyrinth applications of the STAMP benchmark suite, with varying local transactional loads, and number of threads. However, this time, assigning transactions to either of abort/rollback mechanisms entails exploring all possible choices.

We show results of select experiments in Figure 3-4, in particular, of applications with less transactional delay. The outcome of these experiments are quite promising as we were able to obtain instances where the hybrid implementation performs better than all the three- $CaPR^{+}$, $CaPR^{+}(Abort)$, and $RSTM$. Although the speed-up witnessed is marginal, the fact that it performs better than other implementations for 0.1ms delay demonstrates the feasibility of the hybrid approach.

A drawback of this static implementation is that it requires the user to identify and tag transactions to run with abort/rollback. The task of choosing the optimal combination is not trivial, due to which brute-force search had to be used in our experiments. Moreover, contention in the application may not be determined/known to the user a priori. In order to address the same, we need to provide an additional instrumentation mechanism which measures the contention. One way to achieve this is to keep track of the average number of conflicts occurring in the execution per unit time. This information could be used by the transactions to determine dynamically the mechanism to which it should subscribe: abort or partial rollback. This dynamic implementation provides additional flexibility to dynamically switch a transaction from partial rollback to abort and back, even during its execution. The switch from partial rollback to abort can be done anytime during its execution. However, switching a transaction from abort to partial rollback, during the transaction's lifetime, may lead to inconsistent results or in the worst case lead to program crash(rolling back to an intermediate point without restoring the state may lead to segmentation fault). This can be avoided by performing the switch only after the transaction has completely aborted, and before it continues with its execution.

3.4 Discussion

The comprehensive analysis of Abort and Partial Rollback carried out in this work has led to the conclusion that partial rollback is better-off only for applications with large transactions and high contention among threads, with upto 1.6x speedup achieved for *CaPR+* with respect to RSTM for large transactional delays of the order of 100ms. However, this also suggests that a partial rollback based STM may not be practical for most applications, where most of the transactions are short. This result helped us to arrive at the integrated partial rollback-abort framework, that exploits the advantages of both mechanisms. This hybrid implementation is found to be more performance efficient than all the other implementations considered in this study, and is able to attain improvements upto 8%, even when the transactional delays ar of the order of 0.1ms to 1ms. Although a static implementation of the hybrid approach has been presented in this study, the dynamic implementation is expected to be more efficient and productive, which can dynamically switch between mechanisms based on the length of transactions and the contention. This makes it a good candidate for automatic optimization purposes. We plan to employ machine learning techniques in future to realize this hybrid implementation that can automatically gauge the parameters and switch between mechanisms dynamically, leading to better flexibility and performance.

Chapter 4

A Deadlock-free lock-based synchronization for GPUs

Graphics processor units (GPUs) have traditionally been used for data-parallel or task-parallel applications that involve minimal inter- (thread) block communication due to their ability to efficiently exploit thread level parallelism (TLP). However, as synchronization support was made available gradually, GPUs are increasingly being used for general purpose computation. This is also referred to as General Purpose GPU (GPGPU) Computing. Irregular algorithms, in particular, are receiving considerable attention from the GPU community because of their challenges. Their complex memory accesses and data dependent control-flow patterns make them difficult to parallelize. Moreover, the lack of efficient inter- (thread) block communication support adds to the challenges posed to a programmer.

Threads from different thread blocks communicate via global memory accesses while threads within a thread block communicate through shared memory Since concurrent accesses to global memory may result in data-race, recent GPUs provide inter-block communication support in the form of atomic operations for single 32/64-bit words. These atomic operations can be leveraged to construct fine-grained locks but current lock implementations are either prone to deadlocks or result in inferior hardware utilization.

A GPU is built of multiple streaming multiprocessors (SM), each of which are composed of multiple scalar processors. GPU threads, on the other hand, are organized as blocks, each of which is mapped to a single SM. Threads within a block execute in groups of 32 called a warp such that all the threads in a warp execute the same instruction. Synchronization mechanisms in GPUs can be classified based on the level of this thread hierarchy at which it acts i.e. coarse-grained (thread block), medium-grained (warp), or fine-grained (thread). Global barriers have been implemented by Xiao and Feng [87] to provide coarse-grained synchronization, but they result in performance degradation due to reduced TLP. Compute Unified Device Architecture (CUDA), an application programming interfaces (API) for heteregeneous programming on NVIDIA GPUs, provides medium-grained synchronization support in the form of the barrier function *_syncthreads()*, which has been widely adopted, although the programmer must ensure that it is not used inside a divergent branch. However, there is no explicit support by CUDA for synchronization of threads across different blocks (fine-grained synchronization).

In order to expand the scope of GPGPU computing, it is important to provide adequate fine-grained inter-block synchronization support while also handling concurrency issues that may arise due to fine-grained synchronization. In this work, we discuss different ways in which deadlocks can occur in GPU applications due to the existing synchronization mechanisms. This work focuses on deadlocks due to fine-grained locks: SIMD deadlock, deadlock due to circular locking and alias deadlock. SIMD deadlock occurs due to the SIMT execution paradigm of GPUs. Sarnath [93, 91, 83] has proposed a locking scheme using branch divergence that prevents SIMD deadlocks.However, there is still a possibility of deadlocks due to circular locking in the presence of nested locks. Although deadlocks in GPUs due to circular locking are similar to deadlocks in CPUs, since GPUs are comprised of thousands of cores and are capable of executing millions of threads, it becomes very difficult to debug in the event of a deadlock due to the huge amount of debug data analysis required [88, 89]. It becomes even more difficult for the programmer to verify that for every possible input, the irregular memory access patterns among the concurrent tasks would never result in a deadlock. Thus, providing a deadlock-free locking solution would considerably save the programmers efforts. Xu et al. [84] have proposed a solution based on lock stealing that prevents deadlocks due to circular locking within a warp. However, its limitations make it impractical to be used in irregular applications. First, due to the use of lock-stealing, any changes made by a thread to shared data before a lock is stolen from it, has to be undone using a rollback operation. Moreover, this support has to be provided by the programmer. Second, since inter-warp lock stealing is not allowed, circular locking due to threads from different warps is still possible.

In this work, we have described possible deadlock scenarios in GPUs, and present a formal model for SIMD and circular deadlocks in GPUs. We have presented a novel lock-based deadlock-free synchronization mechanism for GPU architectures that uses the solution based on branch divergence to avoid SIMD deadlock, and at the same time also avoids deadlocks due to circular locking for intra-warp and inter-warp threads. It may be noted that the solutions provided here are very much applicable to circular deadlocks in CPUs as well and differs signifcantly from similar algorithms proposed for classical deadocks. We have also established the correctness of Par-Deadlockfree algorithm. We evaluated the efficacy of our approach by implementing the Delaunay Mesh Refinement using our synchronization mechanisms and comparing it with the performance of the three-phase DMR that uses global barriers, and our performance analysis shows that the overhead is quite reasonable. This work has been published in [86].

This chapter is organized as follows: In the next section, we give a background of the GPU architecture and CUDA, the GPU programming model used in this work. In Section 4.2, we discuss the problem of deadlocks in CPUs and GPUs. In Section 4.2.1, we discuss handling of deadlocks in CPUs, and discuss modelling of classic deadlocks using Resource Allocation Graphs (RAG). In Section 4.2.2, we provide a detailed account of the existing synchronization mechanisms in GPUs and the deadlock conditions arising from their use. Existing solutions to overcome some of these deadlocks in GPUs have been discussed in Section 4.2.3. In Section 4.3, we present a formal model for SIMD and circular deadlocks in GPUs. In Section 4.4, we present our efficient, deadlock-free, and livelock-free lock implementation - Par-Deadlockfree. The performance of the proposed algorithm and comparison with existing mechanisms is discussed in Section 4.5.

4.1 Background

4.1.1 Overview of GPU architecture

This work primarily focuses on NVIDIA GPUs, however, the fundamental principles and certain characteristics overlaps with other architectures as well. A GPU is built of multiple streaming multiprocessors (SM), which are composed of multiple scalar processors. For example, NVIDIA TITAN X, a GPU based on the pascal architecture comprises of 28 SMs, each SM comprising 128 cores, making for a total of 3584 cores. GPU threads are organized as blocks, each of which is mapped to a single SM. Threads within a block execute in groups of 32 called a warp. Each warp on a multiprocessor executes in lock-step.Since threads in a warp execute one common instruction at a time (but with different data), lockstep execution can be thought of as having an inherent barrier after every instruction. This is also referred to as the Single Instruction Multiple Thread (SIMT) execution model.

Every multiprocessor of a GPU device has a on-chip shared memory, that is available to all the threads within a block, and registers that are private to each thread. Shared memory is equivalent to a user-managed cache: The application has to explicitly allocate and access it [36]. The global memory and texture memory are available to all the threads of all blocks, but are off-chip and hence have a high latency than the shared memory.

A multiprocessor may handle several warps at a given time and switching between these warps is done to abstract memory latencies and pipeline stalls. This is also referred to as latency hiding. For maximum efficiency, all the threads in a warp must agree on their execution path. However, this may not be always possible due to branch divergence. Branch divergence occurs in the presence of a data-dependent conditional branch. The warp then serially performs execution of each branch path taken, halting threads not on that path. The threads converge after the completion of every path. However, only threads in the same warp face branch divergence; threads in different warps execute independently regardless of their respective execution paths. Branch divergence in GPUs can also be explained using the predicated lock-step execution semantics presented by Collingbourne et al. [106]: an if condition is evaluated into a Boolean variable e, the if statement is removed and the statements inside the if block are predicated by the associated Boolean variable. A predicated statement is now a no-op (no-operation) if e evaluates to false.

4.1.2 CUDA programming model

In order to execute general purpose programs on heterogeneous systems built of multicore CPUs and GPUs, various application programming interfaces (APIs) are available such as CUDA [36], OpenCL [76, 77] and OpenACC [78]. OpenCL ((Open Computing Language)) is a programming language standard that enables the programmer to structure an applications computation as kernels, enabling higher utilization of parallelism naturally available in hardware constructs. However, it is hard to rewrite the existing codes from scratch using these programming models. OpenACC is an open standard that enables the application developers to migrate their applications to heterogeneous systems mainly by inserting compiler directives into the existing code.

CUDA [36] is a high performance programming platform specially designed for NVIDIA GPUs. It extends the C programming model to provide functions for parallel execution of thousands of threads onto a GPU device from a host CPU. Instructions to be executed by each thread is called a kernel and is defined using the `global` specifier. An example of a kernel in CUDA is shown below:

```
// Kernel definition
__global__ void printElement(float* d_vector)
{
int index = threadIdx.x;
printf("Thread ID: %d value: %d",index,d_vector[index]);
}
```

CUDA allows a large number of threads to be created that execute a GPU kernel. These threads are grouped into blocks and blocks make up a grid. A kernel is invoked by the host, which requires the number of blocks in the grid and threads per block, also referred to as ‘kernel execution parameters’, to be specified by the programmer. However, modern architectures are capable of launching kernels from within a kernel, without any CPU involvement. This is also referred to as Dynamic Parallelism. Kernel invocation is illustrated in the following example:

```
int main()
{
...
// Kernel invocation with one block of N threads
printElement<<<1, N>>>(d_vector);
...
}
```

General computation on the GPU proceeds as follows: memory is allocated on the GPU by the host, data is copied from CPU memory to the allocated GPU memory, the kernel is invoked and finally, after the execution of kernel, the output data is copied back to the host.

Figure 4-1 illustrates the GPU architecture and CUDA programming model. It shows a kernel execution where two thread blocks are assigned to each streaming multiprocessor.

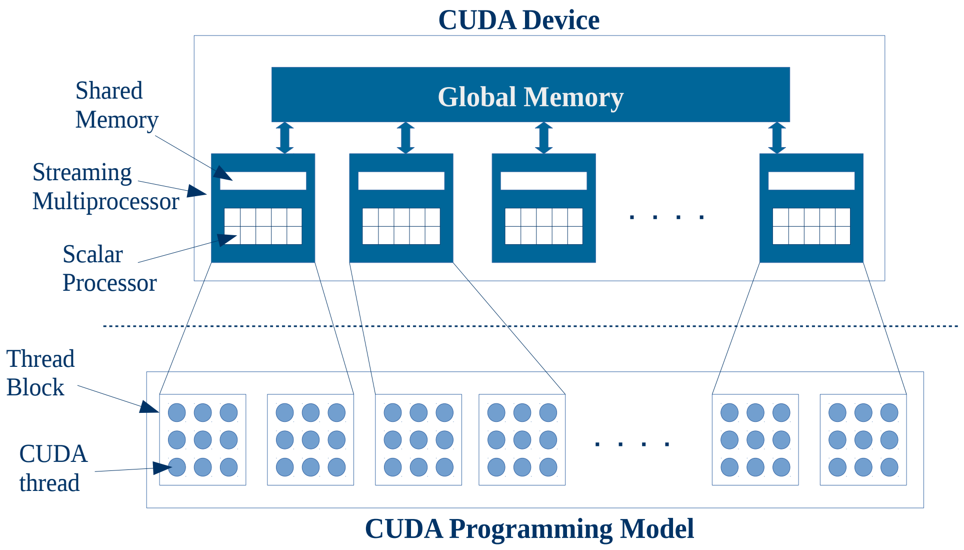


Figure 4-1: The GPU architecture and CUDA programming model

4.2 Deadlocks

In this section, we shall describe the problem of deadlocks in classic architectures as well as GPUs.

4.2.1 Classic Deadlocks

Deadlock is the situation in which multiple processes in a system are blocked forever because of requirements that cannot be satisfied. Coffman et al. [107] listed the conditions that must simultaneously hold, for a deadlock to occur in multi-process systems. They are mutual exclusion, hold and wait, no preemption, and circular wait. Deadlock-free mechanisms have conventionally been devised for CPUs. Classically, deadlocks are handled broadly on the following lines:

1. deadlock prevention: ensure that at least one of Coffman's conditions for deadlocks does not hold true.

2. deadlock avoidance: the system has a priori resource usage information. For

each resource request, it then decides dynamically if the request can be granted, so as to avoid a deadlock, and

3. deadlock detection and resolution: allows the system to enter a deadlock state and then takes action to recover.

Deadlock prevention algorithms eliminate one of Coffman's conditions to prevent deadlocks. Hold-and-Wait condition can be handled by requiring a process to request and be allocated all its resources at start. This requirement may result in poor resource utilization and starvation. The first algorithm given by us - Prev-Deadlockfree is based on this idea. Imposing a total ordering on all resources eliminates circular deadlock condition, whereas allowing processes to pre-empt resources from other processes eliminates the no-preemption condition. Wait-Die and Wound-Wait are two algorithms based on timestamps given by RosenKrantz et al. [117]. With Wait-die, an older process waiting for a younger process is allowed to wait while a younger process waiting for an older process is killed. Wound-Wait gives priority to an older process, and pre-empts the younger process if the younger process is in possession of a resource needed by the older process.

Banker's algorithm is a deadlock avoidance algorithm by Dijkstra [118] that requires a safety check to be performed on every resource request, to decide whether allocation of the resource leads to a safe state, which incurs significant overhead. Further Banker's algorithm is centralized as opposed to our solution, adding to the overhead.

Modeling Classic Deadlocks

Holt [122] showed that classic deadlocks can be modelled using Resource Allocation Graphs (RAG), a directed graph, G(V, E).V consists of two partitions: P = {P_1, P_2, P_3, ..., P_n} is the set of active processes, and R = {R_1, R_2, R_3, ..., R_m} is the set of resources. Processes are represented by a circular node, while the resources are represented using a square. Similarly, E also consists of two partitions: request edges ($P_i \rightarrow R_j$) to indicate that P_i is waiting for the resource R_j, while assignment edges ($R_j \rightarrow P_i$) indicate that R_j has been allocated to P_i. A cycle in RAG indicates presence of a deadlock.

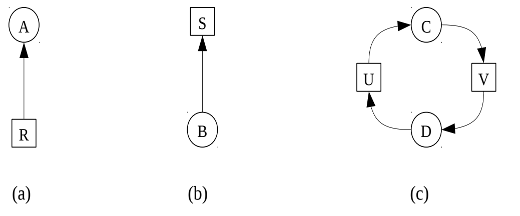


Figure 4-2: Resource allocation graphs: (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Figure 4-2(a) is a simple RAG to show that resource R has been allocated to process A. Figure 4-2(b) shows that process B is waiting for resource S. A deadlock involving two processes and two resources is shown in Figure 4-2(c), where Process C has been allocated resource U and is waiting for resource T. While Process D has been allocated T and is waiting for C to release U. Thus both C and D are waiting for each other to release an object, leading to a deadlock.

4.2.2 Deadlock Scenarios in GPUs

As far as GPUs are concerned, their distinct characteristics result in new challenges, which renders the existing solutions for CPUs unworthy. As opposed to CPUs, GPUs are built on SIMD architecture. They execute thousands of threads concurrently and operate on very large data sets. All of these factors amplify the challenges of programming for GPUs. Due to this reason, deadlocks may arise because of the programming model used for programming GPUs. SIMD deadlocks is an example of a deadlock in GPUs that arises due to the SIMD execution paradigm. The use of fine-grained locks is necessary to prevent data races without degrading performance significantly. However, the use of locks increases the complexity of writing and debugging applications, and may result in deadlocks as well. Deadlocks in GPUs do not get the necessary attention for the simple reason that the usual GPU applications involve minimal synchronization. The increasing attention to GPGPU computing has led to introduction of a variety of synchronization mechanisms for GPUs, which in turn, allow more applications to be computed on GPUs like irregular algorithms. However, their complex memory accesses and control flow patterns increase the possibility of deadlocks. The GPU solutions should address the challenges discussed in Section 4.4, and at the same time should also lead to good hardware utilization. Before describing the deadlock scenarios in GPUs, we discuss various synchronization mechanisms in GPUs.

GPU Synchronizations

Synchronization mechanisms in GPUs can be classified into three granularities based on the level of thread hierarchy at which it acts:

1. coarse-grained

   Coarse-grained synchronization is used when synchronization is required among thread blocks. It can be easily realized through CPU by terminating the current kernel, calling cudaThreadSynchronize function(blocks the host code execution until the device has completed all preceding requested tasks), and re-launching a new kernel. However, the involvement of CPU results in significant performance degradation. Global barrier was first implemented by Volkov and Demmel [123] that used the off-chip GPU memory to enable synchronization across GPU thread blocks. Two global barriers have been implemented by Xiao and Feng [87] that are based on syncthreads (a CUDA command to synchronize warps in the same thread block). One is lock-based that uses a mutex variable and an atomic add operation to implement the global barrier, while the other is lock-free that uses arrays and eliminates the need for mutex and atomic operations. All of these mechanisms avoid the kernel launch overhead by allowing synchronization without the involvement of CPU.

2. medium-grained

   Medium-grained synchronization is used to achieve synchronization among warps in a thread block. GPU provides hardware support for medium-grained warp barriers in the form of the barrier function _syncthreads() that acts as a barrier for all warps in a block, and has been widely adopted.

3. fine-grained

   Fine-grained synchronization is used to achieve synchronization among threads in thread blocks. It is realized using atomic operations, that are guaranteed to be performed without interference from other threads. Fine-grained locks can be built using basic atomic operations provided by CUDA Programming model.

We now illustrate the possible deadlock scenarios in GPUs due to the use of these synchronization mechanisms, and also discuss the existing solutions to overcome some of them.

Deadlock Scenario due to global barrier usage

Using global barriers may lead to deadlocks in the following way. Since execution of thread blocks in CUDA is non-preemptive, the unscheduled blocks will not be able to preempt the active blocks, and thus, never reach the barrier. This is because execution of thread blocks in CUDA is non-preemptive, due to which the unscheduled blocks will not be able to preempt the active blocks, and thus never reach the barrier. Thus, the active blocks would keep on waiting for the unscheduled blocks to reach the barrier, leading to a deadlock.

Illustration

Consider an example where five thread blocks with Ids B_1 - B_5 are created for executing a kernel (consisting of a global barrier) on a GPU with only four SMs. At any point of time, only four blocks can be scheduled to the SMs. Without any loss of generality, let us assume that blocks B_1 - B_4 have been scheduled. On reaching the global barrier, blocks B_1 - B_4 wait for B_5 to reach the barrier. However, since B_5 is not able to preempt blocks B_1 - B_4, it is never scheduled and is not able to reach the barrier, leading to a deadlock.

Deadlock Scenario due to _syncthreads()

Calls to _syncthreads() get compiled into the bar.sync instruction of the PTX (parallel thread execution) instruction set architecture. All the threads in a warp are stalled until the barrier completes, and the arrival count for the barrier is incremented by the warp size [94]. The synchronization completes when the arrival count has been incremented by each of the warps in the thread block. If none of the threads of a warp take the execution path that consists of the _syncthreads() function, the arrival count is never incremented for that warp, leading to a deadlock.

Illustration

Consider the following code snippet of a GPU kernel with kernel execution configuration parameters set to 1 block, 64 threads.

```
if(threadIdx.x > 32 ){
    //some computation;
    __syncthreads();
    //some computation;
}
//some more computation;
```

Consider 2 threads T_{16} (with ID=16, and belonging to warp W_0) and T_{40} (with ID=40, and belonging to warp W_1). Clearly, thread T_{16} does not satisfy the if condition and hence does not execute the if block. Due to this, arrival count is never incremented for this warp (W_0). As a result, T_{40} and other threads (of W_1) keep on waiting for this arrival count to be incremented, and do not progress. Whereas threads of W_0 wait at the end of if block for the threads of W_1 to continue with the lockstep execution, leading to a deadlock.

Alias Deadlock

As opposed to global memory that is visible to all the threads, shared memory is visible only to threads on a block. However, being on-chip, it has lesser memory access latency and higher bandwidth as compared to global memory. Shared memory provides fast atomic access using a module called lock unit, as described by Coon et al. [95](See Figure 4-3). The lock unit comprises of flags called lock bits that indicate the current status of a lock in the corresponding location in Storage resource. Multiple locations in the Storage resource are aliased to a single lock bit, using hashing. This aliasing leads to the possibility of a deadlock when a thread in possession of the lock bit of a memory location tries to fetch an aliased location of the same lock bit.

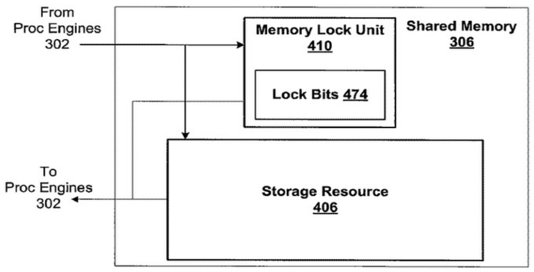


Figure 4-3: Shared Memory lock unit

SIMD Deadlocks

Consider a simple lock implementation using atomic operations as given below:

```
bool lock(Mutex *mutex){
    if (atomicCAS(mutex, 0, 1) == 1)
        return false;
    else
        return true;
}

void unlock(Mutex *mutex){
    atomicExch(mutex, 0);
}
```

Now, using spinlock to realize locking on GPUs may lead to a deadlock. Consider the locking scheme of Table 4.1.

Table 4.1: Scheme 1

T1	T2

`while(!  lock (&x ));`
`Critical Section`
`unlock (&x );`
	`while (!  lock (&x ));`
	`Critical Section`
	`unlock (&x );`

Here, threads T_1 and T_2 within a warp compete for a spinlock, implemented using atomicCAS, i.e atomic compare-and-swap operation. Since T_1 and T_2 belong to the same warp, they execute in lockstep due to the SIMD execution. Now, if T_1 obtains the lock, it exits the while loop and waits at the start of critical section for T_2 to converge, while T_2 spins forever, waiting for T_1 to release the lock; eventually leading to a deadlock. This is referred to as a SIMD deadlock.

Deadlock Scenario due to circular locking

Deadlocks due to circular locking may happen due to threads within a warp, and across warps as well. We discuss both the scenarios here.

Circular Deadlock within a warp

In the presence of nested locks, there is a possibility of deadlocks due to circular wait. Consider the example from Table 4.2, and suppose there is just one warp for execution (when kernel execution parameter is given as one block of say, 16 threads).

Table 4.2: Example to illustrate circular deadlock within a warp

	Thread T_1	**Thread T_2**
1)	lock(a1);	lock(a2);
2)	//critical section 1	//critical section 1
3)	lock(a2);	lock(a1);
4)	//critical section 2	//critical section 2
5)	unlock(a2);	unlock(a1);
6)	unlock(a1);	unlock(a2);
7)	lock(a3);	lock(a5);
8)	lock(a4);	lock(a6);
9)	//critical section 3	//critical section 3
10)	unlock(a4);	unlock(a6);
11)	unlock(a3);	unlock(a5)

Assuming this code is free of SIMD deadlocks, let us see what happens when this is executed. Since Threads T_1 and T_2 belong to the same warp, they execute in lockstep. T_1 acquires lock on a1, while T_2 acquires lock on a2. This leads to a deadlock at Line 3 (Table 4.2), as both of them are now waiting for each other to release the locks already possessed by them.

Circular Deadlock across warps

Having discussed the case of circular deadlock involving threads within a warp, let us now see what happens when a thread from a different warp requests a common resource. With multiple warps, a similar situation may arise. Consider the example shown in Table 4.3.

Table 4.3: Example to illustrate circular deadlock across warps

	Thread T_i	**Thread** T_j	**Thread** T_k
1	lock(a1);	lock(a2);	lock(a3)
2)	lock(a2);	lock(a3);	lock(a1)
3)	//critical section	//critical section	//critical section
4)	unlock(a2);	unlock(a3);	unlock(a1);
5)	unlock(a1);	unlock(a2);	unlock(a3);

Suppose Threads T_i and T_j belong to the same warp W_m, and Thread T_k belongs to a different warp W_n, irrespective of whether W_n is in the same or different thread block as W_m. It is to be noted that since T_k belongs to a different warp, its operations may be arbitrarily interleaved with the operations of T_i and T_j, both of which execute in lockstep. Consider an execution where T_i acquires lock on a1 simultaneously with T_j on a2. Now, if T_k acquires lock on a3 before T_j, it results in a circular deadlock since all of T_i, T_j and T_k are then waiting for each other to release their locks.

4.2.3 Overcoming Deadlocks in GPUs

In this section, we discuss solutions for overcoming deadlock scenarios described above.

Deadlock due to global barrier usage

A simple solution to prevent this deadlock is to restrict the number of blocks to be less than or equal to the number of SM, as described by Xiao and Feng [87]. To ensure two or more blocks are not scheduled on the same SM, each block needs to be allocated all available shared memory on an SM. Although this solution prevents deadlock, it limits the scalability significantly, since it limits the number of thread blocks (and hence threads) that can be used in a kernel.

Deadlock due to _syncthreads()

Deadlocks due _syncthreads() can be avoided by ensuring that _syncthreads() does not appear inside conditional codes, as stated in the CUDA programming guide [36].

Alias Deadlocks

Li et. al [92] have provided a solution that mandates the programmer to responsibly handle this deadlock by preventing illegal access to locked locations in the main storage, by using the lock bits appropriately as the lock unit is not configured to track ownership of locks.

SIMD Deadlocks

SIMD deadlocks can be prevented using Scheme 2, first proposed by Sarnath [93, 91, 83], which uses a divergence mechanism to separate the execution paths of the conflicting threads within a warp. This mechanism uses nested while and if statements, and a local variable 'done' to ensure that execution paths of the divergent threads (belonging to the same warp) are serialized to prevent them from blocking each other.

```
/*Scheme 2: Try-lock*/
  done = false;
  while (done == false){
      if (lock(&mutex)){
          critical section...
```

```
            unlock(&mutex);
            done = true;
        }
    }
```

In scheme 2, divergence can happen in two cases: one due to while statement and another due to if condition. Due to divergence in Scheme 2 as a result of the if condition, the execution paths are separated into two: one for the threads that have acquired the respective locks, and another for the failed threads, which wait for the successful threads to complete the critical section and unlock the locks. At this point, all the threads converge and continue their execution, thereby avoiding the SIMD deadlock.

Consider the example shown in Table 4.1. To prevent SIMD deadlock, we transform this code to acquire locks using Scheme 2. Suppose one of the threads, say T_1, acquires the lock. This leads to a divergence due to if statement. In accordance with predicated semantics explained earlier, T_1 executes the critical section and releases the lock, while T_2 remains idle all this while. At the end of if block, the threads converge again and continue execution, but this time T_1 evaluates the while condition to false. T_2 continues execution, acquires the lock, executes the critical section and releases the lock, while T_1 remains idle. Both the threads now converge again and continue their execution.

Circular Deadlocks

There are two cases of circular deadlocks:

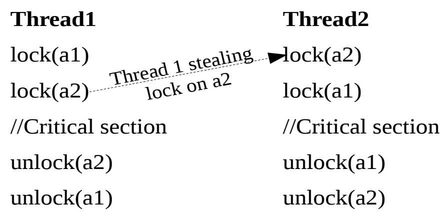
1. due to threads within a warp

2. due to threads across warps

We discuss only the first case here, for which a solution exists.

Circular Deadlocks within a warp

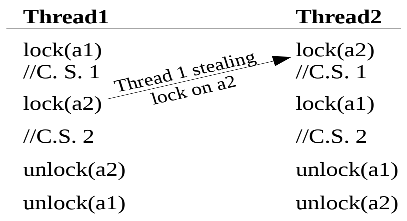
Xu et al. [84] have proposed a pre-emptive solution for GPUs based on lock stealing that prevents deadlocks due to circular locking within a warp. Consider the example in Figure 4-4.



```
Thread1                Thread2
lock(a1)               lock(a2)
lock(a2)               lock(a1)
//Critical section     //Critical section
unlock(a2)             unlock(a1)
unlock(a1)             unlock(a2)
```

Figure 4-4: Example to illustrate lock stealing.

With their solution, circular deadlock is handled by letting the thread with smaller ID, in this case Thread 1, to steal the lock on a2. Now that Thread 1 has acquired all the required locks, it can proceed to completion, and then Thread 2 proceeds. However, their solution is wrong because due to the use of lock-stealing, a thread may have made updates to shared data before a lock is stolen from it. This is illustrated using the example in Figure 4-5. In this example, Threads 1 and 2 have acquired



```
Thread1                Thread2
lock(a1)               lock(a2)
//C. S. 1              //C.S. 1
lock(a2)               lock(a1)
//C.S. 2               //C.S. 2
unlock(a2)             unlock(a1)
unlock(a1)             unlock(a2)
```

Figure 4-5: Example to illustrate the problem with lock stealing.

locks on a1 and a2, and have executed their respective critical sections and written to a1 and a2. Similar to the previous example, when Thread 1 requests for lock on a2 (Line 3), that is held by Thread 2, Thread 1 with the smaller ID gets to steal the lock (on a2) from Thread 2. However, since Thread 2 has finished executing the critical section, it may have made updates to a2. For consistency, these updates need to be rolled back. Their solution requires the programmer to handle this rollback operation by writing a procedure to undo the shared updates already performed by the thread. Although their solution is able to handle circular deadlock within a warp, circular locking due to threads across different warps is still possible since inter-warp lock stealing is not allowed.

Table 4.4: Summary of various deadlock scenarios and their solutions

Synchronization Granularity	**Deadlock Scenario**	**Possible Solution**
Coarse-grained	Using global barriers	no. of blocks <= no. of SMs
medium-grained	Using _syncthreads() in conditional code	No use of _syncthreads() in conditional code
Fine-grained	Alias Deadlock	use lock bits appropriately
	SIMD Deadlock	Use Scheme 2
	Circular Locking within a warp	Lock stealing
	Circular Locking across warps	No existing solutions

Table 4.4 summarizes the various deadlock scenarios and their possible solutions discussed in this section. There exists no satisfactory solution in the literature that handles circular deadlocks in GPUs due to threads across different warps. In this work, we provide an integrated solution to handle circular deadlocks of both the cases discussed above, i.e., within a warp and across warps. Before discussing our solution to the above two cases, let us formally model the circular deadlocks in GPUs.

4.3 Modelling Circular Deadlocks

In this section, we shall discuss graph based modelling of circular deadlocks in GPUs. First, let us model deadlocks within a warp which follows SIMD execution.

Due to the SIMD execution, the Resource Allocation Graphs discussed in Section 4.2.1 cannot be used to model deadlocks in GPUs in the existing form. To overcome this limitation, we provide an extension to Resource Allocation Graphs that enables modelling of deadlocks in GPUs. We refer to them as Extended Resource Allocation Graphs, (ERAG). Similar to RAG, an ERAG also is a directed graph G(V, E), but with an additional node type and an edge type. Threads (as opposed to processes in a CPU) in a GPU are represented as circles, and resources as squares. Now since all the threads in a warp execute in lockstep, we model a warp as a composite node, comprising of one node for each thread in the warp. A warp is shown as a composite node in Figure 4-6.

Request edges and assignment edges are same as before. Due to lockstep execution, if one of the threads, say T_i, in a warp W_k gets blocked (eg. waiting for a lock), all the other threads in the warp also get blocked, waiting for T_i. This waiting is captured as an edge between T_i and every other T_j in W_k. We refer to this edge ($T_j \rightarrow T_i$) as wait edge. It is to be noted that unlike request and assignment edges that connect a thread and a resource, wait edges connect two threads (of the same warp). Having described all the possible nodes and edges, we now show the ERAGs for a) thread holding a resource, b) thread requesting a resource, and c) SIMD deadlock in Figure 4-6.

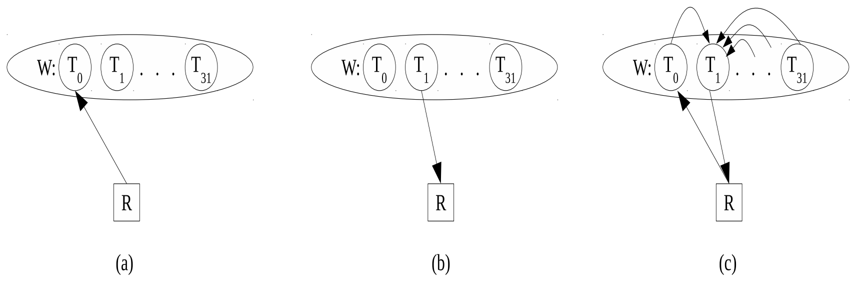


Figure 4-6: Extended resource allocation graphs: (a) Holding a resource. (b) Requesting a resource. (c) SIMD Deadlock.

In Figure 4-6 (c), thread T_0 is allocated the resource R, and when T_1 requests R, it waits for T_0 to release R. However, T_0 cannot proceed as it is waiting for T_1 (due to lockstep execution), leading to a deadlock. This deadlock is captured by ERAG as a cycle involving the sole request and assignment edges and the wait edge between T_0 and T_1. An important point to be noted is that if locks are acquired using Scheme 2, the wait edges that result in SIMD deadlock do not appear on the ERAG. Thus SIMD deadlock will never occur, when used with Scheme 2.

Having discussed ERAG, let us now model circular deadlocks within a warp. Assuming there are no SIMD deadlocks (these can be handled using Scheme 2, and hence no wait edges appear in ERAG), the ERAG for circular deadlock for the example in Table 4.2 is shown in Figure 4-7. Here, threads T_1 and T_2 have acquired locks on a1 and a2 respectively (represented by assignment edges in the ERAG). Now, when T_1

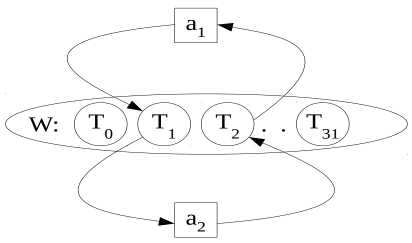


Figure 4-7: Extended resource allocation graph of circular deadlock (at Line 3 of Table 4.2).

requests lock on a2 (represented by a request edge), it gets blocked since a2 is held by T_2. Similarly, when T_2 requests lock on a1, a cycle is formed, resulting in a deadlock. As other threads do not participate in the deadlock, their assignment edges have been obscured in the ERAG for simplicity.

This deadlock will not immediately prevent other threads of the same warp from executing (assuming they attempt to acquire different locks). However, as these threads converge after the execution of Line 6, they wait for T_1 and T_2 to converge, which are deadlocked. This stalls the execution of all the other threads in the warp as well. This blocking is due to lockstep execution, and is captured as a wait edge. The ERAG at this point is shown below (Figure 4-8):

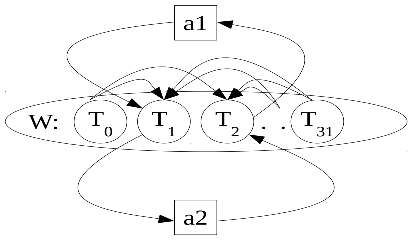


Figure 4-8: Extended resource allocation graph of circular deadlock (at Line 7 of Table 4.2).

It is to be noted that since these wait edges appear at the end of the nested interval (Lines 1 - 6), the threads (except T_1 and T_2) have not requested any of the locks, and consequently will never be part of a circular deadlock (and SIMD deadlock as well), since there will be no outgoing request edges or incoming assignment edges from these thread nodes.

Modelling circular deadlocks due to threads across warps

Circular deadlock due to threads across warps can be modeled as an ERAG as shown in Figure 4-9.

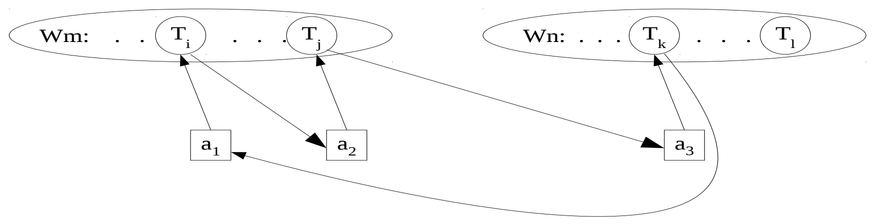


Figure 4-9: Extended resource allocation graphs of circular deadlock with threads from different warps.

Figure 4-9 shows the ERAG for the example shown in Table 4.3. This ERAG illustrates a state where thread T_i is waiting for T_j to release lock on a2, T_j is waiting for T_k to release a3, while T_k is waiting for T_i to release a1. Thus, none of them can make forward progress, resulting in a deadlock.

Assuming there is no SIMD deadlocks in each of the warps (which can be handled using Scheme 2), the intra-warp wait edges are not present in the ERAG. Due to this reason, the modelling becomes the same for both the cases, and hence the same solution can be used to handle circular deadlocks both within and across warps. In the following section, we shall discuss our solution explicitly for circular deadlocks within and across warps.

4.4 Deadlock-free lock algorithms for GPUs

As discussed in the previous section, synchronizations in GPUs have to be handled carefully due to the challenges introduced by the SIMD paradigm. Our objective is to arrive at lock-based GPU synchronizations that offer the following properties:

1. Deadlock-freedom

2. Livelock-freedom

3. Efficiency

In this section, we present 'Prev-Deadlockfree', a naive deadlock-free lock implementation, and then systematically arrive at a more efficient deadlock-free lock implementation, referred to as 'Par-Deadlockfree'.

Consider the example in Table 4.2. Suppose thread T_1 has acquired a1, and before it requests a2, some other thread, say T_2 (may or may not belong to the same warp) has acquired a2. In this case, T_1 is required to wait for T_2 to release a2. Now, when T_2 requests a1, it leads to a deadlock. In this example, there are two nested intervals: lines 1-6 form interval I_1, and lines 7-11 form interval I_2. Towards this, we identify all the nested intervals in the given source code. This is important because after the completion of an interval, it can no more be party to a deadlock. So we limit our scope to intervals, rather than the entire code. We say that a1 is the 'outermost resource' in the interval I_1. All the resources contained within the lock and unlock operations of the outermost resource are said to be the 'dependent resources' of the outermost resource, regardless of whether there is further level of nesting of locks.

To implement the example of Table 4.2, the first step is to transform the calls to lock and unlock operations as per Scheme 2, to prevent SIMD deadlocks. With Scheme 2, the actual code of Thread 1 and Thread 2 (for brevity, code for only Lines 1-6 is shown) would transform[a] to as given below:

```
/* THREAD 1     */               /* THREAD 2 */
done1 = false;                    done1 = false;
while (done1==false){             while (done1 == false){
  if (lock(a1)){                    if (lock(a2)){
    /*C.S. 1*/                        /*C.S. 1*/
    done2 = false;                    done2 = false;
    while (done2==false){             while (done2 == false){
```

[a]this transformation can be automated easily using macros.

```
      if (lock(a2)){                      if (lock(a1)){
        /*C.S. 2*/                          /*C.S. 2*/
        unlock(a2);                         unlock(a1);
        done2 = true;                       done2 = true;
      }                                   }
    }                                   }
    unlock(a1);                         unlock(a2);
    done1 = true;                       done1 = true;
  }                                   }
}                                   }
```

Although this code prevents SIMD deadlock, it is still susceptible to circular deadlocks.

We now present our solution to this circular deadlock problem. We first give a naive lock algorithm, Prev-Deadlockfree, that is deadlock-free but limits the amount of parallelism. We improve upon this algorithm and incrementally arrive at our lock algorithm through two attempts - Par-Synchronize1 and Par-Synchronize2. In pursuit of more efficiency, we arrive at Par-Synchronize1 that offers more parallelism than Prev-Deadlockfree, but is still prone to deadlocks. Par-Synchronize2 introduces an additional check to prevent deadlocks, but is prone to livelocks. This leads us to Par-Deadlockfree that is deadlock-free, livelock-free and also offers more parallelism as compared to Prev-Deadlockfree.

4.4.1 Prev-Deadlockfree: A Naive Deadlock Prevention Algorithm

This algorithm is based on deadlock prevention. It prevents deadlock by ensuring that the 'hold and wait' condition never holds true. This is realized by requiring threads to acquire all the resources in an interval before they can start execution of the interval. This algorithm looks at the resource usage information of the interval under consideration, and grants access to an outermost resource only if it is able to acquire its lock and the locks of all its dependent resources. The algorithm assumes that the dependent set for each outermost resource in an interval is known. The algorithm is given in Algorithm 1.

Algorithm 1 *Prev-Deadlockfree*

```
1:  procedure LOCK(x)
2:      if x is a dependent resource then
3:          return true
4:      else                                          ▷ x is an outermost resourcel
5:          status ← Acquire locks on all resources(outermost + dependent)
6:          if status = failure then
7:              release all locks, return false
8:          else                                      ▷ status = success
9:              return true
10: procedure UNLOCK(x)
11:     release lock on x
end
```

On calling the `lock` function for an outermost resource x, locks are acquired on x and its dependent resources (line 5, Algorithm 1). In case of failure in acquiring lock of any of the resources, the (already) acquired locks are revoked (line 7). Unlock function is trivial, the lock is released, irrespective of whether it is an outermost resource or a dependent resource.

Illustration of Prev-Deadlockfree

Consider the following example (Table 4.5) comprising of threads T_1, T_2 and T_3, executing in parallel, where Threads T_1 and T_2 belong to same warp, where as T_3 is from a different warp.

Table 4.6 gives the execution time-line for the above example. Suppose the current state of lock vector (for a1, a2, a3, a4) is lock: (0, 0, 0, 0).

Table 4.5: Example to illustrate Prev-Deadlockfree

T_1	T_2	T_3
lock(a1);	lock(a3);	lock(a2);
critical section 1;	critical section 1;	critical section 1;
lock(a2);	lock(a4);	lock(a4);
critical section 2;	critical section 2;	critical section 2;
unlock(a2);	unlock(a4);	unlock(a4);
unlock(a1);	unlock(a3);	unlock(a2);
non-critical section;	non-critical section;	non-critical section;

Here, Threads T_1, T_2 and T_3 require locks on a1, a3 and a2 respectively. Thread T_1 can acquire lock on a1 only if it can set a1.lock=1 and a2.lock=1. Similarly for T_2 and T_3. Suppose T_3 is able to acquire lock on a2, which means it would have set a2.lock=1 and a4.lock=1, which restrains T_1 and T_2 from obtaining their locks. T_3 continues its execution, while T_1 and T_2 keep checking for their respective locks. As soon as a4 is released by T_3, T_2 is able to acquire lock on a3. This leads to branch divergence(due to if condition of Scheme 2) and T_2 continues its execution, while T_1 is disabled. After a3 is unlocked by T_2, both T_1 and T_2 converge. Now, again, T_1 and T_2 diverge(this time, due to while condition of Scheme 2) and T_1 continues its execution while T_2 is disabled. After a1 is unlocked by T_1, both threads converge and execute to completion.

Limitation

Like most of the synchronization techniques, Prev-Deadlockfree also involves certain trade-offs. First, it needs to figure out the dependent set of an interval in advance. Another drawback is due to the requirement of acquiring all the dependent locks beforehand, that results in hardware underutilization and minimal throughput.

To overcome the latter drawback, we present our improved solution to address circular deadlocks in GPUs that is more efficient than Prev-Deadlockfree.

Table 4.6: Execution Time-line for the example in Table 4.5

Thread T_1	**Thread** T_2	**Thread** T_3
lock(a1)→ false	**lock(a3)**→ false	**lock(a2)**→true
lock(a1)→ false	**lock(a3)**→ false	critical section 1;
lock(a1)→ false	**lock(a3)**→ false	**lock(a4)**→true
lock(a1)→ false	**lock(a3)**→ false	critical section 2;
lock(a1)→ false	**lock(a3)**→ false	**unlock(a4)**
lock(a1)→ false	**lock(a3)**→ true	**unlock(a2)**
	critical section 1;	non critical section;
	lock(a4)→ true	
	critical section 2;	
	unlock(a4)	
	unlock(a3)	
lock(a1)→ true		
critical section 1;		
lock(a2)→ true		
critical section 2;		
unlock(a2)		
unlock(a1)		
non critical section;	non critical section;	

4.4.2 Par-Deadlockfree: An efficient Deadlock Prevention Algorithm

To overcome the limitations of Prev-Deadlockfree, we propose Par-Deadlockfree, an improved lock algorithm that is also based on deadlock prevention. For a better understanding, we arrive at Par-Deadlockfree by step-by-step development through Par-Synchronize1 and Par-Synchronize2, enhancements over Prev-Deadlockfree that offer more parallelism, but are still prone to deadlocks/livelocks.

Par-Synchronize1

In order to improve the hardware utilization, instead of acquiring locks on all the dependent resources, Par-Synchronize1 maintains a boolean called 'bool' for each dependent resource, which indicates if that resource is a dependent resource for some (already acquired) resource. Then, a request for lock on an outermost resource x by thread T_i is granted under the following conditions:

1. The lock for x is available. (line 9)
2. The value of bool for x is 0 (x=1 implies there's some thread T_j for which x is the dependent resource. So the grant of x to T_i would cause T_j to wait for T_i to release x.). (line 8)
3. All of x's dependent resources have their bool value 0 (ensures none of its dependent resources are required by other threads (already dependent resource).). (lines 10-11)

While condition 1 is obvious, conditions 2 and 3 just require that the outermost resource x and its dependent resources are themselves not the dependent resources of any other resources. We capture the three conditions in Par-Synchronize1.

Algorithm 2 *Par-Synchronize1*

```
 1: procedure LOCK(x)
 2:     if x is a dependent resource then
 3:         if atomicCAS(x.lock, 0, 1)=1 then
 4:             return true
 5:         else
 6:             return false
 7:     else                                        ▷ x is an outermost resource
 8:         if x.bool = 0 then
 9:             if atomicCAS(x.lock,0,1) then
10:                 status ← Acquire bool on all dependent resources;
11:                 if status = failure then
12:                     release all bools
13:                     release lock on x
14:                     return false
15:                 return true
```

```
16:             else
17:                 return false
18:         else
19:             return false
20: procedure UNLOCK(x)
21:     $x.lock = 0$
22:     if ( x is a dependent resource) then
23:         $x.bool = 0$
end
```

If x is a dependent resource, the implementation just checks for the availability of lock on x. If x is an outermost resource, it checks for the 3 conditions listed above. In case the 3rd condition is not satisfied, it revokes the bool value to 0 for all the resources (line 12), revokes the lock on x and returns false(lines 13-14). If all the 3 conditions are satisfied it returns true(line 15).

The third condition may lead to livelock because if condition 3 fails, the algorithm requires Ti to revoke the bool values to 0 for each element in x's dependent set. We avoid this by keeping the resources in the dependent set in a sorted sequence.

Note: Par-Synchronize1 offers more parallelism than Prev-Deadlockfree. We show this using the following example.

Table 4.7: Example to illustrate efficiency of Par-Sychronize1

Thread T_1	**Thread** T_2
lock(a1);	lock(a2);
//critical section 1	//critical section 1
lock(a2);	lock(a1);
//critical section 2	//critical section 2
unlock(a2);	unlock(a1);
unlock(a1);	unlock(a2);

Suppose the state at the beginning of its execution for locks and bools on (a1,a2,a3,a4) is - lock:(0,1,0,0), bool:(0,0,0,0).

This indicates that the lock on resource a2 is already acquired by some other thread. Prev-Deadlockfree would not admit this request (lock(a1)) as it requires lock on a2 also to be acquired. However, with Par-Synchronize1, we just check the bool value for a2, find it to be 0, and hence satisfies all the 3 conditions. Thus, request for a1 by T_1 can be granted.

Limitation

Evidently, Par-Synchronize1 provides far better hardware utilization and throughput in comparison with Prev-Deadlockfree. However, a major drawback of this algorithm is its failure to avoid deadlock in certain scenarios.

Consider the example shown in Table 4.3 where two threads from different warps execute in parallel. Here, lock(a1) is called from Thread T_1, which finds a1.bool to be zero (step 9, Algorithm 2). As a result, T_1 will proceed further. However, if at the same time, T_2 is in step 11, this will find a1.bool to 0, change it to 1 and secure the lock on a2, resulting in deadlock.

Par-Synchronize2

To resolve this issue, two approaches are proposed:

1. Par-Synchronize2 introduces an additional check of the outermost resource's bool.

2. The dependent set is modified to include the outermost resource in the set, to ensure prevention of deadlock.

Before granting a lock, a check is made whether bool of the outermost resource is set to 0 or not. If it is 1 (`x.bool=1`), granting of the lock may result in deadlock since the outermost resource is also in dependent set of some other thread. Therefore, all the bools that are acquired by present thread are released. This approach successfully resolves the issue of deadlock. However, due to the rollback strategy adopted, it may result in livelock as two threads may wait infinitely for bools of their respective dependent resources. This shortcoming is illustrated using the example shown in Table 4.3. Suppose Threads T_1 and T_2 belong to the same warp. Since initially `a1.bool = 0` and `a2.bool = 0`, both the threads will proceed further and try making bools of their dependent set to 1 i.e T_1 will make `a2.bool = 1` and T_2 will make `a1.bool = 1`. The additional checking step of T_1 will find `a1.bool = 1` and T_2 will find `a2.bool = 1`, hence both the thread will perform the rollback and repeat the same steps, resulting in livelock.

We now discuss our Par-Deadlockfree algorithm that is deadlock-free, livelock-free and at the same time offers more efficiency than Prev-Deadlockfree.

Par-Deadlockfree

Par-Deadlockfree provides a better approach to solve the deadlock/livelock issues of Par-Synchronize1 and Par-Synchronize2 by performing certain modification in the dependent set and overcoming the limitations of livelocks. Like majority of its atomic operation based synchronization peers, Par-Synchronize1 falls into deadlock due to simultaneous modification on shared data (i.e. bool vector) by multiple threads. In order to solve this, `LOCK` function of Par-Deadlockfree sets the outermost resource's bool in addition to its lock to 1, thus ensuring that no other thread will attempt to acquire this resource.

After acquiring all the bools of modified dependent set (Step 9, Algorithm 3) and lock on the resource (Step 13), bool of the outermost resource is released (Step 14). In case any of the bools or lock is already acquired, rollback takes place i.e. bool of every resource in modified dependent set is made 0 (Step 11/17). Hence, lock will not be granted. To obtain lock on a dependent resource, it just checks the availability of lock, and if acquired, it sets its bool to 0.

Intuitively, `UNLOCK` function changes lock of the resource to 0. The algorithm is given in Algorithm 3.

Algorithm 3 *Par-Deadlockfree*

```
1: procedure LOCK(x)
2:     if x is a dependent resource then
3:         if atomicCAS(x.lock,0,1)=1 then
4:             release bool on x, return true
5:         else
6:             return false
7:     else                                        ▷ x is an outermost resource
8:         status ← Acquire bool on all resources(outermost + dependent)
9:         if status = failure then
10:            release all bools, return false
11:        else                                    ▷ status = success
12:            if atomicCAS(x.lock,0,1) = 0 then
13:                release bool on x
14:                return true
15:            else                                ▷ x is already locked
16:                release all bools
17:                return false
18: procedure UNLOCK(x)
19:     release lock on x
end
```

Illustration of Par-Deadlockfree

We illustrate Algorithm 3 using the example below (Table 4.8):

In this example, Threads T_1 and T_2 belong to the same warp, while Thread T_3 belongs to a different warp. Suppose T_1 and T_3 acquire their respective locks and proceed to their `critical section 1`, however, T_2 cannot acquire `a3.lock` since `a3.bool` is set to 1 by T_3. Since T_1 and T_2 are in the same warp, T_2 is disabled till both the threads agree on the same execution path due to branch divergence (as a

Table 4.8: Example to illustrate Par-Deadlockfree

Thread T_1	**Thread** T_2	**Thread** T_3
lock(a1);	lock(a3);	lock(a2);
critical section 1;	critical section 1;	critical section 1;
lock(a2);	lock(a4);	lock(a4);
critical section 2;	critical section 2;	critical section 2;
unlock(a2);	unlock(a4);	unlock(a4);
unlock(a1);	unlock(a3);	unlock(a2);
non-critical section;	non-critical section;	non-critical section;

result of failed if condition of Scheme 2). T_1 tries to acquire `a2.lock` which is held by T_3, hence T_1 will keep on checking for `a2.lock` and finally will be granted `a2.lock` only after `a2.lock` is released by T_3. Eventually, after T_1 releases `a1.lock`, both T_1 and T_2 converge. Now, T_2 can proceed further, assuming `a3.bool` has been set to 0 by some other thread in the meanwhile.

Consider another example where T_3 tries to acquire locks for a2 and then a1 instead of a4. Clearly, if both T_1 and T_3 are allowed to acquire a1 and a2 respectively, it will lead to a deadlock. However, Par-Deadlockfree does not permit this as the thread which acquires its outermost resource first would have changed its dependent set's bool to 1, thereby restricting the other thread to acquire its outermost resource. A formal proof of deadlock freedom for the more general case involving n threads is discussed below.

4.4.3 Proof of Deadlock Freedom

Theorem: Par-Deadlockfree is deadlock-free.

Proof (by contradiction):

Let us assume that a deadlock occurs, involving n threads, leading to cyclic wait. Suppose the cycle formed is as follows:

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow ... \rightarrow T_n \rightarrow T_1$.

Consider Thread T_1. For T_1,

1. $T_n \rightarrow T_1 \Rightarrow T_1$ has already acquired some resource, say a_1.

2. $T_1 \rightarrow T_2 \Rightarrow T_1$ is waiting for some resource, say a_2, to be released by thread T_2.

This could happen only if $\mathit{lock}(a_1)$ precedes $\mathit{lock}(a_2)$ in thread T_1. This implies that at the point when $\mathit{lock}(a_1)$ is acquired, $\mathit{bool}(a_2) = 1$, regardless of whether a_1 is an outermost resource for the interval in consideration. This is true because the algorithm acquires bool for all dependent resources at the time of acquiring lock of outermost resource.

Since $\mathit{bool}(a_2) = 1$, thread T_2 could not have acquired $\mathit{lock}(a_2)$ after $\mathit{lock}(a_1)$.

$$\Rightarrow T_2.\mathit{lock}(a_2) \rightarrow T_1.\mathit{lock}(a_1).$$

Similarly, we have

$$\Rightarrow T_3.\mathit{lock}(a_3) \rightarrow T_2.\mathit{lock}(a_2)$$
$$.$$
$$.$$
$$\Rightarrow T_n.\mathit{lock}(a_n) \rightarrow T_{n-1}.\mathit{lock}(a_{n-1})$$
$$\Rightarrow T_1.\mathit{lock}(a_1) \rightarrow T_n.\mathit{lock}(a_n).$$

By transitivity, we get $T_1.\mathit{lock}(a_1) \rightarrow T_1.\mathit{lock}(a_1)$, which is not possible, thereby contradicting our assumption, thereby establishing that Par-Deadlockfree is deadlock-free.

4.5 Performance Evaluation

We first analyze the performance of the Prev-Deadlockfree and the proposed Par-Deadlockfree lock algorithms. To achieve this, we model the kernel as an interval similar to that of the example shown in Table 4.2 and carry out the experiments. Further, to compare the performance of our algorithm with existing works, we use the Delaunay Mesh Refinement application implemented using different synchronization mechanisms. For a Delaunay triangulation, none of the points lie inside the circumcircle of any of its triangle. Delaunay Mesh Refinement imposes certain quality constraints on such a Delaunay triangulation or mesh (For example no triangle has an angle of less than 30 degrees). It involves iteratively processing the mesh for triangles that violate the quality constraint (also called bad triangles), until no more bad triangles are left.

Experimental Framework

The experiments were performed on a compute node with following configuration: 2 Intel(R) Xeon(R) CPU E5-2620@ 2.00GHz with 24 cores, 2 Tesla K40 GPUs, 32 GB RAM and 30 GB SWAP. The Tesla K40 GPU comprises 2880 cores running at 745 MHz, with 12 GB memory. We used CentOS 7.1.1503 with gcc-4.8.3 and CUDA-7.0 for compilation.

4.5.1 Prev-Deadlockfree vs Par-Deadlockfree

For comparing the performance of Prev-Deadlockfree and Par-Deadlockfree, we have written a micro-benchmark application whose kernel is modeled as an interval similar to that of the example shown in Table 4.2. Arbitrary delays are introduced into the kernel that resemble critical sections. We then perform experiments by varying the number of blocks and observe the execution time and locking failures for the proposed algorithms. The delay, number of objects and block size are kept constant throughout the experiments. The results are shown in Figure 4-10. The execution times are in logarithmic scale to address skewness in values of observed data.

It is evident from Figure 4-10 that Par-Deadlockfree performs better than Prev-Deadlockfree. Par-Deadlockfree executes faster in all cases except for 1 block, in which case Prev-Deadlockfree is faster by up to 10%. The maximum improvement is achieved by Par-Deadlockfree with 8 blocks, with performance gain of 32% been observed, while the average gain observed over all cases for Par-Deadlockfree is 5%.

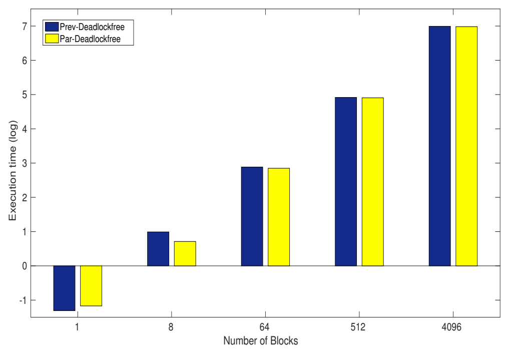


Figure 4-10: Comparative evaluation of Prev-Deadlockfree and Par-Deadlockfree

4.5.2 Comparative Evaluation of Par-Deadlockfree with existing works

For comparison of Par-Deadlockfree with existing works, we have modified the DMR implementation of the LonestarGPU project by Burtscher, Nasre and Pingali [110] to use Par-Deadlockfree for synchronization, and compare its performance with

1. the original DMR implementation of LonestarGPU that uses the 3-phase conflict resolution scheme proposed by Nasre, Burtscher and Pingali [109].

2. DMR implemented using the lock stealing based synchronization mechanism proposed by Yunlong et al. [84]. The pseudo code for the GPU implementation of DMR using Par-Deadlockfree is shown in Algorithm 4.

Algorithm 4 *Delaunay Mesh Refinement*

```
1: procedure REFINEMESH(Mesh mesh)
2:     count ← checkTriangles(mesh)
3:     while count > 0 do
4:         refine(wl)
5:         count ← checkTriangles(mesh)
6: procedure REFINE(worklist wl)
7:     for each triangle t in wl do
8:         if if t is bad and not deleted then
```

```
9:          $S \leftarrow$ Create cavity
10:         $S = S \bigcup t$
11:         Sort $S$ and lock all triangles in $S$ in order
12:         if locking of any triangle fails then
13:             unlock all triangles locked
14:             release all bools and continue
15:         create new cavity by re-triangulating
16:         delete triangles in old cavity from mesh
17:         add triangles in new cavity to mesh
18:         unlock all triangles locked
end
```

The benchmark inputs for DMR are used from the Galois project by Keshav et al.[108]. Each experiment is repeated 5 times, and a graph is plotted with the arithmetic mean of the observed execution times. The execution times are in logarithmic scale to address skewness in values of observed data. The results are shown in Figure 4-11.

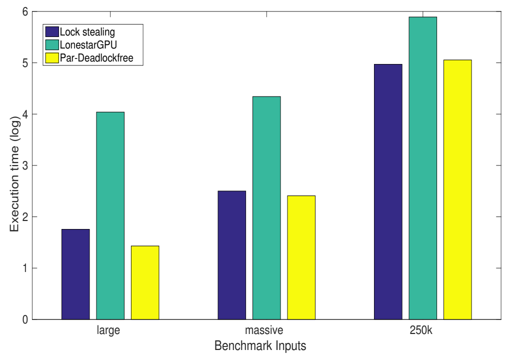


Figure 4-11: Comparative evaluation of DMR implementations

From the results, we see that Par-Deadlockfree outperforms LonestarGPU DMR implementation for all the benchmark inputs with up to 13.5x speedup observed for the smallest input. When compared to the lock stealing implementation, Par-Deadlockfree performs better for smaller inputs, however it lags for the 250k input, although marginally (8%). 250k, the largest test input, consists of a mesh with 0.5 million triangles, of which half of them are bad initially. The marginal performance decline of Par-Deadlockfree can be justified by the fact that it gives an additional guarantee of deadlock-freedom.

4.6 Discussion

Par-Deadlockfree provides a deadlock-free GPU synchronization mechanism that is efficient in-spite of its simplicity. However, an important concern is the difficulty in managing locks in large applications. Improper use of locks may affect the correctness of the application. Moreover, Par-Deadlockfree limits its scope to applications where the resource usage information is known in advance. This limitation can be overcome by using it for optimistic concurrency control mechanisms like Transactional Memory. Transactional Memories (TM) allow pieces of code earmarked as transactions to run atomically by executing them optimistically. With a lazy TM, all the write operations to shared objects are performed on their local copies, and changes to these objects are reflected in the shared memory only when the transaction commits. Thus, TM provides a high level abstraction, where he/she just has to identify the transactions in an application, and the TM implementation takes care of all the synchronization and communication. The implementation may internally use locks.

Par-Deadlockfree is a suitable candidate for the implementation of such a TM for GPUs, since a transaction would precisely know its resource usage information at commit time. Running transactions optimistically with an appropriate contention manager is also expected to offer better efficiency as it would allow maximum transactions with disjoint write-sets to commit. Moreover, the high-level abstraction offered by TMs also increases productivity of the programmer. The characteristics of GPUs would require the usage of appropriate data structures for book-keeping that are efficient and light-weight.

Chapter 5

Exploiting parallelism and recursion to realize performance

Multi-core computers have now emerged as the mainstream computing platform. Graphic Processor Units (GPU), in particular, have evolved into highly parallel many-core processors with high throughput and memory bandwidth that make them suitable for compute-intensive applications, that can be expressed as data-parallel computations. While GPU lends itself to a high level of parallelism, leveraging it optimally is considerably difficult. CUDA allows the programmer to partition the problem into sub-problems, known as blocks, that can be assigned to the different multiprocessors, while each of the sub-problems itself is partitioned and solved in parallel by multiple threads within the multiprocessor. Although CUDA simplifies programming to an extent by abstracting some of the low-level details from the user, the user is still required to specify the execution parameters, keeping in mind the multi-level memory hierarchy and other GPU parameters. The application needs to be tuned to arrive at the optimal execution parameters. Scaling computations on clusters is even more challenging due to the added complexity of multiplicity of nodes, and inter-node communication.

We address this problem using Powerlist, a data structure that enables us to specify parallel algorithms. Powerlist allows us to represent several algorithms concisely because of the use of recursion and also because it avoids explicit indexing of a powerlists elements using the operations defined on them. We demonstrate the expressive power of powerlists through Cooley-Tukey algorithm [100], a widely popular algorithm for computing Fast Fourier Transform (FFT). The cooley-Tukey algorithm is shown below:

```
RECURSIVE-FFT($a$)
$n = a.length$                // $n$ is a power of 2
if $n == 1$
    return $a$
$\omega_n = e^{2\pi i/n}$
$\omega = 1$
$a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} =$ RECURSIVE-FFT($a^{[0]}$)
$y^{[1]} =$ RECURSIVE-FFT($a^{[1]}$)
for $k = 0$ to $n/2 - 1$
    $y_k = y_k^{[0]} + \omega\, y_k^{[1]}$
    $y_{k+(n/2)} = y_k^{[0]} - \omega\, y_k^{[1]}$
    $\omega = \omega\, \omega_n$
return $y$                    // $y$ is assumed to be a column vector
```

Figure 5-1: Cooley-Tukey algorithm for computing FFT [101].

Since Powerlist is able to capture both parallelism and recursion at the same time, it is able to express the above algorithm succinctly as a recursive definition in the powerlist notation as follows:

$$fft. < a >=< a >$$

$$fft.(pq) = fft.p + u * fft.q | fft.p - u * fft.q$$

where, $u =< \omega^0, \omega^1, ..., \omega^{N-1} >$, N is the length of p, ω is the $2 * N$th principal root of 1.

This concise description of the Cooley-Tukey algorithm is possible due to the astraction provided by powerlist. This abstraction can be leveraged to shield the user from the programming complexity resulting from the multi-level memory hierarchy of multi-core architectures.

In this chapter, we show how powerlists can be leveraged on parallel systems comprising of GPUs to enhance both performance and productivity. We first provide a method to arrive at the kernel execution parameters from the GPU characteristics, that results in optimal performance. This approach helps in making the programmer oblivious to the underlying hardware (GPU) parameters, thereby improving performance and productivity considerably, as the same program can now execute efficiently at different GPUs without any additional effort. We then extend this work to allow computations to be scaled across a cluster of GPUs. In particular, we implement a library of powerlist operations that automates the partitioning and merging of subproblems. This library is then used to provide a scheduling algorithm for matrix multiplication across a cluster of GPUs, thereby overcoming the CUDA Basic Linear Algebra Subroutines (cuBLAS) library's limitation to schedule an application across the cluster. This work has been published in [98].

The chapter has been organized as follows: in the next section, we provide a background of the GPU architecture and CUDA programming model as well as a brief overview of the powerlist data structure. In Section 5.2, we present powerlist specifications for the matrix multiplication algorithm. In Section 5.3, we present a powerlist-based approach for GPUs that predicts how threads for an application should be mapped to the GPU cores, based on the underlying hardware parameters. We then extend this work to allow computations to be scaled across a cluster of GPUs in Section 5.4. We demonstrate this high-level framework through SGEMM subroutine of cuBLAS library, wherein the partitioned matrices are automatically scheduled over a GPU cluster. In Section 5.5, we report our observation that powerlists can be used to specify most of Cache-Oblivious and Multicore Oblivious algorithms.

5.1 Background

5.1.1 Characteristics of GPU

The overview of GPU architecture and the CUDA programming model has been discussed in Section 4.1. In this section, we discuss various architectural parameters of GPU that affect the performance of GPU applications. Table 5.1 lists some of these parameters that influence the performance, and the corresponding values for Tesla M2050, the GPU used in our experiments:

Table 5.1: GPU parameters

S.No	GPU Parameter	Denoted by	Value (for Tesla M2050)
1	No. of multiprocessors per GPU	M	14
2	No. of cores per multiprocessor	N	32
3	Max no. of threads per MP	tmp_max	1536
4	Max no. of threads per block	thb_max	1024
5	Max no. of registers per block	rgb_max	32768
6	Max no. of thread blocks per MP	bmp	8
7	Shared memory per block(and MP)	SM_avail	48KB
8	Warp size	w_size	32

The programmer is required to understand these detailed hardware features and its constraints to optimize programs for high performance. The application needs to be tuned for these parameters for optimal performance. To simplify this task, we show how powerlists can be used to optimally predict the kernel execution configuration parameters.

We now give an overview of the powerlist data structure.

5.1.2 Powerlists

Powerlist was introduced by Jay Misra in 1994 [103]. Powerlist is a data structure that enables us to specify parallel algorithms. It highlights the role of both parallelism and recursion, and in fact many recursive algorithms like FFT, Batcher Sort, Prefix-Sum etc. can be described succinctly using powerlists [33], [105].

Formally, a powerlist can be described as a list whose length is equal to a power of two. A list whose length is not a power of two can also be represented as a powerlist by padding it up with (junk) elements. The elements of a powerlist are enclosed within angular brackets. The base powerlist is a list containing one element, $\langle a \rangle$, also called singleton. Larger powerlists can be composed from two smaller powerlists of equal length and same type by using the zip($\bowtie$) and tie($|$) operations defined below:

$p|q$ (tie operation): concatenates the elements of the p and q powerlists.
$p \bowtie q$ (zip operation): interleaves the elements of p and q.

Example:
1) $\langle 2,3 \rangle | \langle 1,4 \rangle = \langle 2,3,1,4 \rangle$
2) $\langle 2,3 \rangle \bowtie \langle 1,4 \rangle = \langle 2,1,3,4 \rangle$

Clearly, the composition of powerlists using tie and zip operations results in a larger powerlist of double the length of the component powerlists. Conversely, any (non-singleton) powerlist can be expressed as the zip or tie of two smaller powerlists of half its length. This way, zip and tie operations can also be used as deconstructors to break down the lists. The commutativity law is the only law that relates $|$ and $\bowtie$, and is given by,

$$(p|q) \bowtie (u|v) = (p \bowtie u)|(q \bowtie v)$$

Multi-Dimensional Array

An n-dimensional array can be represented by a powerlist of depth n-1. For example, we represent a 2-dimensional matrix, having m rows and n columns, by a powerlist having m elements, each of which is a powerlist containing n elements, one for each column. This necessitates application of operations to elements of any specified dimension. This is made possible by defining operations $|', \bowtie', |", \bowtie$ " etc. where $|'$

applies the operation | to the powerlists in dimension 1. Similarly, |" applies | to powerlists in dimension 2 and so on, and likewise for the zip operation.

Example:

1) $<< 2 >, < 3 >> \mid << 1 >, < 4 >> = << 2 >, < 3 >, < 1 >, < 4 >>$

2) $<< 2 >, < 3 >> \mid' << 1 >, < 4 >> = << 2, 1 >, < 3, 4 >>$

Powerlists can also be mapped onto hypercubes [33], [105]. An n-dimensional hypercube can be viewed as a graph with 2^n nodes where each node is represented by an n-bit binary string. So, if we denote an element of a powerlist of length 2^n by a n-bit binary string, representing its position in the powerlist, it can be mapped onto a hypercube of size 2^n to the node with the same label. Several hypercube algorithms have been listed in [33] and [105].

Misra has given the powerlist representation of matrices in [103], and given algorithms for FFT, Batcher-sort, Prefix-Sum etc. In [33], J. Kornerup has given an algorithm for matrix multiplication using powerarrays, a higher dimension structure than powerlists. However, this leads to complex recursive specification of matrix multiplication. So, in the following section, we provide two algorithms for matrix multiplication using the powerlist notation. [34] gives a quadtree representation for matrices, and gives an algorithm for matrix multiplication. It is a divide and conquer algorithm, partitioning the matrices into blocks, and expressing the result as a sum of products of these smaller blocks. Whereas with powerlists, the subproblems are independent from one another, so that they can be computed in parallel, and no consolidation is required. Another advantage of using powerlist is that it completely exposes the inherent concurrency in the application and allows us to have a concise functional description of parallel programs. Moreover, since index notations are not used, they allow us to work at a high level of abstraction. Thus, powerlist representation becomes very relevant in the context of computing matrix multiplication for large matrices, and for other computations which are divide and conquer in nature.

In the following section, we demonstrate how Powerlists can be used to specify computations.

5.2 Specifying Matrix Multiplication using Powerlists

For simplicity, we demonstrate specification of computations using powerlists through matrix multiplication.

Consider a matrix A. Since we want to represent it in the powerlist notation, we require its length to be $n = 2^k$. Although this appears to be a serious limitation, it can be handled easily, by zero-padding the matrices to obtain the required dimensions, without affecting the results significantly. It could be represented in the powerlist notation as a powerlist of powerlists, where each of the row or column forms a powerlist of scalar objects. For example, the first row of the matrix A is given by the powerlist, $p = \langle a_{1,1}, a_{1,2}, \cdots, a_{1,n} \rangle$.
Then, matrix A is given by the powerlist,

$$A = \langle \langle a_{1,1}, a_{1,2}, \cdots, a_{1,n} \rangle, \langle a_{2,1}, a_{2,2}, \cdots, a_{2,n} \rangle, ..., \\ \langle a_{n,1}, a_{n,2}, \cdots, a_{n,n} \rangle \rangle$$

Notation: A matrix here is represented as a powerlist of powerlists. For operations, quite often, it is necessary to use the dimensions of the matrices. For this purpose, we need to use different operators to indicate the partitioning or merging of partitioned matrices. In the case of 2-dimensions, $|$ denotes the concatenation of powerlists, and $|'$ represents concatenation of powerlists of powerlists. For example,
1) $\langle \langle a \rangle, \langle b \rangle \rangle | \langle \langle c \rangle, \langle d \rangle \rangle = \langle \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle \rangle$
2) $\langle \langle a \rangle, \langle b \rangle \rangle |' \langle \langle c \rangle, \langle d \rangle \rangle = \langle \langle a, c \rangle, \langle b, d \rangle \rangle$

We can interpret the above as follows: $|$ increases the dimensions of the matrix by increasing the number of rows, while $|'$ increases the number of columns.

Matrix Multiplication: Algorithm 1

Let A and B be the input matrices. Then, the multiplication of A and B is given by $M(A, B)$ where M is defined below through rules 1, 2, 3 and 4 given below:

Case A: Matrix A and B are 1×1 matrix, i.e., $A = \langle\langle x\rangle\rangle$, and $B = \langle\langle y\rangle\rangle$. Then,

$$M(\langle\langle x\rangle\rangle, \langle\langle y\rangle\rangle) = \langle\langle x \times y\rangle\rangle \tag{1}$$

Here, $\langle\langle x\rangle\rangle$ and $\langle\langle y\rangle\rangle$ are singleton elements (or a 1×1) matrix and $\times$ denotes the classical multiplication of numbers.

Case B: A and B consist of more than one row and column. Let $A = r|s$, and $B = u|v$; Then

$$M(r \mid s, u \mid' v) = (M(r, u) \mid' M(r, v)) \mid (M(s, u) \mid' M(s, v)) \tag{2}$$

Rule 2 partitions the powerlists corresponding to the two matrices recursively and terminates when matrices reduce to matrices consisting of one row or one column. An important thing to note is how Rule 2 is able to precisely capture the partitioning of the A, B and C matrices. Matrix A is partitioned horizontally into 2 blocks r and s while matrix B is partitioned vertically into u and v, thereby giving 4 blocks of matrix C, as given in the RHS of Rule 2.

This gives us a clear idea about how at any level, the subproblems are independent of each other, and hence can be computed in parallel.

Case C: Each matrix consists of only one row(or column) but more than one element. i.e., $A = m|'n$, and $B = p|'q$. Then,

$$M(m|'n, p|q) = S(M(m, p)\ ,\ M(n, q)) \tag{3}$$

At this point, row i and column j obtained could be further partitioned recursively using rule 3 until we get singletons. S() captures the addition operation to evaluate the sum of products.

We now define the rule,

$$S(\langle\langle x\rangle\rangle, \langle\langle y\rangle\rangle) = \langle\langle x + y\rangle\rangle \tag{4}$$

Rule 4 computes the sum of two singleton elements x and y.
Altogether, the computation of matrix multiplication can be captured by using powerlists through the four rules described above.
For illustration, let us compute $M(r_1, c_1)$ from above that computes the product of one row (of A) and one column (of B) to give the corresponding element of C ($M(r_1, c_1)$ itself is obtained after recursive application of Rule 2 that partitions A into rows and B into as many columns).
Let $r_1 = \langle\langle a_1, a_2, a_3, a_4\rangle\rangle$ and $c_1 = \langle\langle b_1\rangle, \langle b_2\rangle, \langle b_3\rangle, \langle b_4\rangle\rangle$

Then,
$M(\mathrm{r}_1\ , c_1) = \mathrm{M}(\langle\langle a_1, a_2, a_3, a_4\rangle\rangle, \langle\langle b_1\rangle, \langle b_2\rangle, \langle b_3\rangle, \langle b_4\rangle\rangle)$
$= M(\langle\langle a_1, a_2\rangle\rangle|'\langle\langle a_3, a_4\rangle\rangle, \langle\langle b_1\rangle, \langle b_2\rangle\rangle|\langle\langle b_3\rangle, \langle b_4\rangle\rangle)$
$\xrightarrow{rule\ 3}$

$S(M(\langle\langle a_1, a_2\rangle\rangle, \langle\langle b_1\rangle, \langle b_2\rangle\rangle), M(\langle\langle a_3, a_4\rangle\rangle, \langle\langle b_3\rangle, \langle b_4\rangle\rangle))$
$\xrightarrow{rule\ 3}$

$S(S(M(\langle\langle a_1\rangle\rangle, \langle\langle b_1\rangle\rangle), M(\langle\langle a_2\rangle, \langle\langle b_2\rangle\rangle)),$

$S(M(\langle\langle a_3\rangle\rangle, \langle\langle b_3\rangle\rangle), M(\langle\langle a_4\rangle\rangle, \langle\langle b_4\rangle\rangle)))$

Now, the powerlists consists of only singleton elements or matrices of size 1×1, and hence Rule 3 cannot be applied further. Instead rule 1 is now applied to get,

$$M(\mathrm{r}_1\ , c_1) = \mathrm{S}(\mathrm{S}(\langle\langle a_1 \times b_1\rangle\rangle, \langle\langle a_2 \times b_2\rangle\rangle)))\ ,$$

$$S(\langle\langle a_3 \times b_3\rangle\rangle, \langle\langle a_4 \times b_4\rangle\rangle)))$$

$\xrightarrow{rule\ 4}$

$$S(\langle\langle a_1 \times b_1 + a_2 \times b_2\rangle\rangle, \langle\langle a_3 \times b_3 + a_4 \times b_4\rangle\rangle))$$

$\xrightarrow{rule\ 4}$

$$\langle\langle a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3 + a_4 \times b_4\rangle\rangle$$

We have shown how the computation of matrix multiplication can be captured by using powerlists. We have seen through illustrations how partitioning is achieved by using the tie and zip operations as deconstructors, that are defined on powerlists. Similarly, the merging can be achieved by using the same tie and zip operations as constructors for combining the partial results.

Matrix Multiplication: Algorithm 2

In the partitioning scheme discussed above, we apply Rule 2 recursively until we obtain single row/column, and then Rule 3 is applied to partition the row/column. However, it is also possible to apply Rule 3 to partition the matrices rather than just rows/columns. This leads us to an efficient but more complicated partitioning scheme of blocked matrix multiplication [36]. We now show how the same rules discussed above could be utilized to arrive at the partitioning scheme of blocked matrix multiplication. Rule 2 partitions matrix A horizontally and B vertically, while Rule 3 partitions A vertically and B horizontally. Thus, we represent this partitioning scheme by integrating Rule 2 and Rule 3, so as to partition A and B matrices along both the dimensions, and is given by Rule 5 below:

$$\begin{aligned}
M(A,B) = M(A_1|A_2, B_1|'B_2) = \\
M([(A_{11}|'A_{12})|(A_{21}|'A_{22})], [(B_{11}|B_{12})|'(B_{21}|B_{22})]) \\
= (S(M(A_{11},B_{11}), M(A_{12},B_{12}))|' \\
S(M(A_{11},B_{21}), M(A_{12},B_{22})))| \\
(S(M(A_{21},B_{11}), M(A_{22},B_{12}))|' \\
S(M(A_{21},B_{21}), M(A_{22},B_{22}))) \quad (5)
\end{aligned}$$

Comparison of the algorithms

Algorithm 2 saves a lot of global memory bandwidth, while also allows us to take advantage of the faster shared memory for reducing the number of accesses to global memory. However, for our purpose, Algorithm 1 is preferred, as we utilize it just for partitioning the input matrices, while the actual computation is performed by the efficient cuBLAS library. The advantage is that the partitioning scheme of Algorithm 1 allows us to partition the problem into independent subproblems, where each of the subproblems is computed using cuBLAS. The resultant product matrices thus obtained just needs to be arranged using $|$ and $|'$ operations to obtain the final product. Where as Algorithm 2 clearly involves further consolidation of the product matrices using Rule 5. For this reason, we use Algorithm 1 in our method for computing matrix multiplication.

In the next section, we show how this algorithm can be exploited for realizing matrix multiplication on GPUs.

5.3 Realizing Matrix Multiplication keeping in view the GPU Parameters

While multicore computers enable us to handle various parts of an application in parallel, writing programs to extract maximum possible performance is hard. In particular, Graphics Processing Unit (GPU) architecture imposes additional programming constraints. Although the CUDA programming model abstracts these constraints, making the kernel code independent of the GPU parameters like number of available multiprocessors etc, the programmer is still required to decompose the problem aptly into a grid of blocks. The number of blocks per grid and the number of threads per block, both of which together (along with the dimensions) constitute the execution configuration parameters, play an important role in determining the performance of the application. This is because these parameters directly affect latency hiding (occupancy) and resource utilization. It is also to be noted that these parameters also depend on resource constraints (register and shared memory). So it becomes necessary for the programmer to identify the suitable execution configuration parameters for the program for which the best performance is obtained, which are dependent on the GPU parameters. Some of the work on providing guidelines to CUDA programmers have been listed in [35], [32], [31].

In this work, we demonstrate how, using powerlists, the parameters of multi- core architectures and in particular GPUs, can be effectively utilized to predict the optimal kernel execution configuration parameters. Thus, this approach becomes productive from the perspective of the programmer. We now illustrate our approach through matrix multiplication.

5.3.1 Matrix Multiplication Algorithm: An Illustrative Example

Given a multicore system with n cores, a problem could be partitioned manually into subproblems, and assigned to the available n cores. This partitioning is not a trivial problem. Having fine granularity for the blocks facilitates better load balancing and higher parallel efficiency, while having coarse granularity provides better performance in sequential execution on a single core. The key then is to identify the crossover point, and could be easily found if this algorithm is expressed in terms of powerlists. The ability of powerlists to model both parallelism and recursion allows us to recursively divide the problem into subproblems, till we arrive at an optimal granularity keeping in view the various hardware parameters and constraints. And then the subproblems could be solved in parallel by the multiple cores.

With GPUs, we also have several parameters(discussed in the previous section) that make the task of identifying the crossover point more difficult. We provide a method to arrive at this crossover point at which the optimal performance is achieved, keeping in view the available GPU parameters. By automatically computing this crossover point, it becomes possible for us to derive the execution configuration parameters for the GPU kernel from it, thus making it productive from the programmers' point of view, as now he is neither required to understand the underlying GPU features and its constraints in depth, nor is he required to tune the program manually for better performance, thus greatly reducing his effort.

We have used matrix multiplication algorithm for illustration rather than algorithms like FFT etc. (even though the visible results for FFT would be superior) for simplicity of presentation.

Given two n×n matrices A and B, we compute matrix C = A * B.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} * B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{bmatrix} = C = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{bmatrix}$$

Then,

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} * b_{k,j} \qquad (6)$$

The classical naive algorithm is shown in Algorithm 5.

Algorithm 5 *Compute $C = A * B$*

```
1: for $i = 1$ to $n$ do
2:     for $j = 1$ to $n$ do
3:         for $k = 1$ to $n$ do
4:             $c[i][j] = c[i][j] + a[i][k] * b[k][j]$
end
```

The complexity of the algorithm is obviously $O(n^3)$ where n is the dimension of the matrix.

Now, let us consider a multicore system with n^2 cores. Intuitively from Equation(5.4.1), it follows that if we can compute n^2 elements of C in parallel, then we would have computed the matrix, C in linear time. To realize this, we could directly partition the problem into n^2 threads and assign them to the available n^2 cores such that all of the threads evaluate in parallel one element of the resultant matrix, C. But when n is very large, the problem has to be partitioned by the programmer appropriately for the best performance, which is a challenging task as the programmer needs to know how to utilize the hardware resources effectively.

One of the ways to overcome the problem is to partition the problem appropriately and our claim is, that this is possible with the use of powerlists. In the next section, we show how powerlists can be exploited to realize matrix multiplication on GPUs.

5.3.2 Realizing the Matrix Multiplication Algorithm

Given the parameters for a particular GPU, let us see how the matrix multiplication algorithm in powerlist notation, presented in the previous section, can be transformed into a GPU program. As shown above, the algorithm works by dividing the problem recursively by deconstructing the matrices, and then reconstructing them again from the partial results. Mapping this algorithm onto the GPU entails dividing the matrix, C into several blocks, and assigning these blocks to the available multiprocessors which compute the results after loading the corresponding sub-matrices from Matrices A and B, and store them into C.

We now show how we deconstruct the matrices A and B of matrix length 2^n and size denoted by $SIZE(=2^{2n})$ using the rules for matrix mulitplication given in Section 5.2. A is represented as the concatenation of two submatrices r and s, while B as that of submatrices u and v. From Rule 2, we represent $C(=M(A,B))$, in terms of the submatrices $M(r,u), M(r,v), M(s,u)$ and $M(s,v)$. This computation can be captured in a CUDA program by partitioning it into 4 blocks of threads (each consisting of $SIZE/4$ threads), one for each of the 4 submatrices $M(r,u), M(r,v), M(s,u)$ and $M(s,v)$.

So, at the next (second) level of recursion, $M(r,u)$ is again expressed as a function of 4 submatrices, requiring 16 blocks altogether, each of size $SIZE/16 = 2^{2n-4}$. The block length then, is given by the square root of the block size, i.e 2^{n-2}.

One of the ways to improve the performance of a GPU program is to reduce the number of accesses to global memory by loading parts of matrices A and B into the shared memory. This is because shared memory accesses are much faster as compared to global memory, and will be discussed in detail in Section 5.3.2. At this(second) level of recursion, to compute a block, we need to load one submatrix each from A and B of size $blocklength * matrixlength$, thereby requiring memory of size $2 * 2^{n-2} * 2^n = SIZE/2$.

Table 5.2 shows the number of blocks, corresponding block size, block length and shared memory requirement at each of the possible levels of recursion.

Table 5.2: Number of blocks, block size, dimension and shared memory required at the different levels of recursion

Level	**No. of blocks**	**Block size**	**Dimension**	**Shared Memory required**
0	1	$SIZE$	2^n	$2 * SIZE$
1	4	$SIZE/4$	2^{n-1}	$SIZE$
2	16	$SIZE/16$	2^{n-2}	$SIZE/2$
3	64	$SIZE/64$	2^{n-3}	$SIZE/4$
i	4^i	$SIZE/4^i$	2^{n-i}	$SIZE/(2^{i-1})$
n	4^n	1	1	2

Evaluating the Optimal Depth of Recursion

In order to implement the Matrix Multiplication algorithm on a GPU, we need to identify the execution configuration parameters, i.e. the number of blocks and the block size, from which the block and the grid dimensions can be derived. For this, the depth of recursion needs to be identified for which the optimal performance is achieved. In this section, we give a method that attempts to arrive at such an optimal depth of recursion, with respect to the GPU parameters and the constraints.

Let us assume the size of the matrices to be $2^n * 2^n$.

Based on the number of blocks, and the number of threads per block, several cases can be identified, and analyzed individually for their time complexity. These are as follows:

Case 1: Number of blocks $\leq$ M (Number of SMs) and Number of threads per block $\leq$ N
In this case, all the blocks are assigned instantaneously to the SMs, and are computed in parallel, Hence the time taken is determined by the time taken by any SM. Again, by the same argument, all the threads are computed within a single batch, and hence the time taken by any SM is given by the time taken by any of the threads (say O(t)).

Thus,

$$timetaken = O(t) \tag{7}$$

Case 2: Number of blocks $\leq$ M (Number of SMs) and Number of threads per block $\geq$ N

Here, the time taken by any of the SMs is determined by the number of batches in which the threads are processed. Hence,

$$timetaken = \lceil (blockSize/N) \rceil * O(t) \tag{8}$$

Case 3: Number of blocks $\geq$ M (Number of SMs) and Number of threads per block $\leq$ N

Here, as Number of blocks $\geq$ M, all the blocks cannot be processed in parallel. The computation spans over several batches in each of which M blocks are computed in parallel. Hence,

$$timetaken = \lceil (Numberofblocks/M) \rceil * O(t) \tag{9}$$

Case 4: Number of blocks $\geq$ M (Number of SMs) and Number of threads per block $\geq$ N

In this case, several batches of M parallel blocks are computed. Within each block, as the number of threads per block $\geq$ N, the threads also are computed in batches of N threads each. Thus,

$$timetaken = \lceil (Numberofblocks/M) \rceil * \lceil (blockSize/N) \rceil * O(t) \tag{10}$$

The case 4, by itself, is the most general case, as all the expressions of time can be deduced by substituting the values of number of blocks and number of threads per blocks in the above expression. Further, the time taken is also a factor of the number of resident warps (warps resident on a multiprocessor), which itself is a factor of the number of resident blocks (blocks resident on a multiprocessor). This number is very much dependent on the GPU parameters discussed in Section 5.1.1, and can be given by:

$$numWarps = \lceil (numResBlocks * blockSize)/warpSize \rceil \tag{11}$$

where numResBlocks is the number of resident blocks and is given by the following expression that aims to capture the constraints imposed by the GPU:

$$numResBlocks = min(\lfloor Numberofblocks/M \rfloor, \lfloor rgbmax/rgb \rfloor, \lfloor tmpmax/blockSize \rfloor, \lfloor SMavail/SMreqd \rfloor, 8) \quad (12)$$

So, the expression for the time taken is given by,

$$timetaken = (\lceil (Numberofblocks/M) \rceil * \lceil (blockSize/N) \rceil * O(t))/numWarps \quad (13)$$

Further, if the shared memory required per block is more than the available shared memory, the blocks need to be further divided into sub-blocks, which further reduces the execution time by a factor of the block size, which has been discussed in Section 5.3.2 in detail. Then, the time taken is given by,

$$timetaken = (\lceil (Numberofblocks/M) \rceil * \lceil (blockSize/N) \rceil * O(t)) /(numWarps * blockSize) \quad (14)$$

The problem in hand is to identify the optimal depth of recursion (level) at which we will be able to get the most efficient implementation of the matrix multiplication algorithm. Let us assume this level to be i. From Table 5.2, we substitute the values of the number of blocks, number of threads per blocks(which is same as the Block size) for the level i along with the other parameters in Equation 14. Doing this gives us the following expression of time complexity in terms of i.

$$timetaken = \left(\frac{\lceil \frac{2^{2i}}{14} \rceil * \lceil 2^{2n-2i-5} \rceil * 2^{2i}}{\lceil numResBlocks * 2^{2n-2i-5} \rceil * 2^{2n}}\right) \quad (15)$$

And then the optimal value of i can be found by considering this expression of time complexity as a function of i and minimizing this function for the values of i. We now demonstrate how this optimal value of i is used for translation of powerlist specifications into a CUDA program.

```
void MatrixMult(Matrix A, Matrix B, Matrix C){
        ...
// Specifying the dimensions
dim3 dimBlock(2^(n-opt_i), 2^(n-opt_i));
dim3 dimGrid(2^opt_i, 2^opt_i);
// Invoking the kernel
MatrixMultiply<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
        ...
}
```

Figure 5-2: Code snippet of kernel invocation

Using Powerlist Specifications for Translation into a CUDA Program

Given matrices A, B of size $2^n * 2^n$, we wish to derive an efficient CUDA program from the specifications of the matrix multiplication algorithm in the powerlist notation. The most important step is to identify the kernel execution configuration parameters for which the maximum performance is achieved, which requires us to specify the grid and block dimensions. These dimensions are directly dependent on the optimal level of recursion, which we refer to as opt_i. In terms of the rules for matrix multiplication from Section 5.2, opt_i level of recursion means that the Rule 2 is applied opt_i number of times. From Table 5.2, at level opt_i, we have number of blocks= 4^{opt_i}. This corresponds to saying that the matrix multiplication problem is partitioned into 4^{opt_i} blocks and each of these blocks are assigned to the M streaming multiprocessors of the GPU. Section 5.3.2 discusses how we find opt_i. Once the optimal depth of recursion, i is estimated, we can directly compute the values of number of blocks and the block size from Table 5.2, and use them to arrive at the CUDA program for matrix multiplication. Thus, at level opt_i, the block length is 2^{n-i}, so the dimensions of the block are given by $2^{n-i} * 2^{n-i}$ and, the dimensions of the grid is then 2^i*2^i. The invocation of the kernel is then given by the code snippet in Figure 5-2.

When the kernel is launched, the blocks are assigned to the streaming multiprocessors. A streaming multiprocessor further has N cores. The threads within a block leverage this parallelism and each of the threads is responsible for computing only one

Figure 5-3: Code snippet of the matrix multiplication kernel

```
__global__ void MatrixMultiply(Matrix A, Matrix B,
Matrix C){

float val = 0;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

if(row > A.height || col > B.width)
        return;
for (int i = 0; i < 2^n; ++i)
        val = val + A.elements[row * 2^n + i]
                        * B.elements[i * 2^n + col];
C.elements[row * C.width + col] = val;

}
```

element of the matrix C. Hence the kernel function is described by the code snippet in Figure 5-3.

Experimental Results

To evaluate the efficacy of our method, we conducted experiments that involved varying the level of recursion, i for different matrix sizes (square matrices of length = power of 2), and observing the kernel execution times. The experiments were conducted on the Nvidia Tesla M2050, that has 14 multiprocessors, each having 32 cores, letting 448 threads run simultaneously. It has 48KB shared memory and allows 32k registers per block. The results are presented in tables 5.3 and 5.4. The entries with value 'N.A' indicates that the execution configuration parameters corresponding to these entries are invalid because the thread block size is greater than 1024, and hence the kernel failed to launch.

The results show that for matrix sizes greater than 256×256, the best performance is achieved for the block size 1024, the largest possible block size for this GPU. As expected the execution time is much higher for executions with small block sizes.

Table 5.3: Time taken for Matrix Multiplication with global memory(in ms)

		Matrix size					
level	# blocks	8x8	16x16	32x32	64x64	128x128	256x256
0	1	0.0143	0.0204	0.0337	N.A	N.A	N.A
1	4	0.0118	0.0121	0.0199	0.0339	N.A	N.A
2	16	0.0115	0.0120	0.0295	0.0251	0.0544	N.A
3	64	0.0119	0.0123	0.0164	0.0248	0.0523	0.2404
4	256		0.0145	0.0185	0.0346	0.0710	0.3482
5	1024			0.0296	0.0580	0.1379	0.5561
6	4096				0.1419	0.3817	1.0936
7	16384					1.3197	3.9690
8	65536						15.2366

Table 5.4: Time taken for Matrix Multiplication with global memory(in ms)

		Matrix size				
level	# blocks	512x512	1024x1024	2048x2048	4096x4096	8192x8192
4	256	2.0277	N.A	N.A	N.A	N.A
5	1024	2.9280	16.8560	N.A	N.A	N.A
6	4096	4.6038	25.2952	132.9967	N.A	N.A
7	16384	10.2599	36.1883	208.0879	1067.5977	N.A
8	65536	35.4952	97.4814	242.6834	1714.8394	8832.1172
9	262144	133.3535	322.9157	793.2737	2337.0413	17724.4531
10	1048576		1254.0564	3061.1614	6371.4443	22213.8516
11	4194304			10389.6152	24952.7461	51490.3633
12	16777216				97949.1641	200577.1094

Using Shared Memory for Matrix Multiplication

As per the algorithm, computing one block (of block length l) of the matrix C(of matrix length N) requires loading the corresponding submatrices A and B each of size $l * N$. However in practice, each of the threads load 2N elements from the global memory, resulting in $2 * l^2 * N$ loads. Since global memory accesses are very slow, this may leave the threads idle for hundreds of clock cycles, for each access. We may therefore instead utilize the shared memory, which is upto 100 times faster than the global memory(since it is on-chip). Figure 5-4 depicts the memory hierarchy for Fermi architecture [37]. The shared memory is utilized by first loading the full or parts of these submatrices into the shared memory. This results in only $2 * N/l$ loads per thread and $2 * l^2 * (N/l)$ loads per block from the global memory to the shared memory, thereby eliminating redundant accesses to global memory. Now that these submatrices are available for every thread on the block for computing, the threads access the elements of A and B matrices from the shared memory, resulting in faster computation.

However in this case we still need to access the global memory in addition to the shared memory, and it should be the case that as we increase the matrix size, beyond one point, using shared memory should be advantageous. We show this through experiments which show that for matrix sizes 1024x1024 and beyond, using shared memory gives better results.

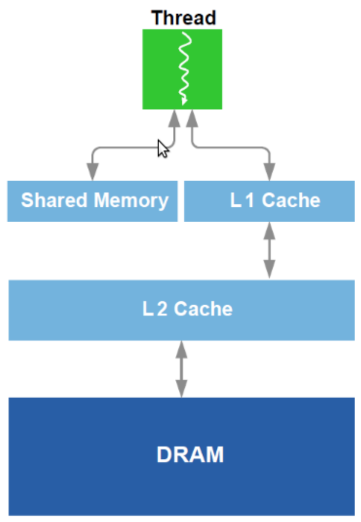


Figure 5-4: GPU Memory Hierarchy

Experimental Results

With the shared memory version of the algorithm, we observed the execution times by varying the level of recursion, i for different matrix sizes, the results of which are given in Tables 5.5 and 5.6.

This implementation involves splitting the submatrices further into sub-blocks if the original submatrices from A and B do not fit into the shared memory, adding up the product of the corresponding sub-blocks from A and B. Comparing it with

Table 5.5: Time taken for Matrix Multiplication with shared memory(in ms)

		Matrix size					
level	# blocks	8x8	16x16	32x32	64x64	128x128	256x256
0	1	0.0484	0.0374	0.0711	N.A	N.A	N.A
1	4	0.0205	0.0149	0.0231	0.0499	N.A	N.A
2	16	0.0211	0.0144	0.0176	0.0452	0.0869	N.A
3	64	0.0226	0.0159	0.0188	0.0317	0.0729	0.3143
4	256		0.0260	0.0339	0.0675	0.1961	0.3253
5	1024			0.0949	0.1809	0.5817	0.5680
6	4096				0.5932	1.2428	5.2327
7	16384					5.2260	9.6159
8	65536						41.3438

Table 5.6: Time taken for Matrix Multiplication with shared memory(in ms)

		Matrix size				
level	# blocks	512x512	1024x1024	2048x2048	4096x4096	8192x8192
4	256	2.0450	N.A	N.A	N.A	N.A
5	1024	2.2680	15.9815	N.A	N.A	N.A
6	4096	4.2161	17.6467	124.9091	N.A	N.A
7	16384	17.8364	33.3412	140.3770	995.6590	N.A
8	65536	100.6139	142.4077	265.6320	1118.0102	7997.3862
9	262144	343.3006	651.3238	1229.2145	2137.0495	9030.3408
10	1048576		3724.3792	5230.4233	9837.1474	17549.9843
11	4194304			31313.9824	47548.2460	79042.2109
12	16777216				304358.1562	1131193.1250

the results in Tables 5.3 and 5.4, we find that the performance with shared memory lags for matrix sizes till 512, but gets better thereafter. Moreover in these cases, the best performance is achieved for block sizes 1024(and block length 32). This may be explained by the fact that with larger block length, the number of global memory accesses are reduced more.

To evaluate the optimality of our approach described in Section 5.3.2, we compare the best results obtained from the experiments, with the results for the values of opt_i (optimal level of recursion) predicted by our approach for the different matrix sizes. Figure 5-5 shows the bar graph, comparing the logarithm of the time taken for the predicted values of i and the values of i for which the best performance is achieved.

The results demonstrate the accuracy of our approach, it is able to predict the

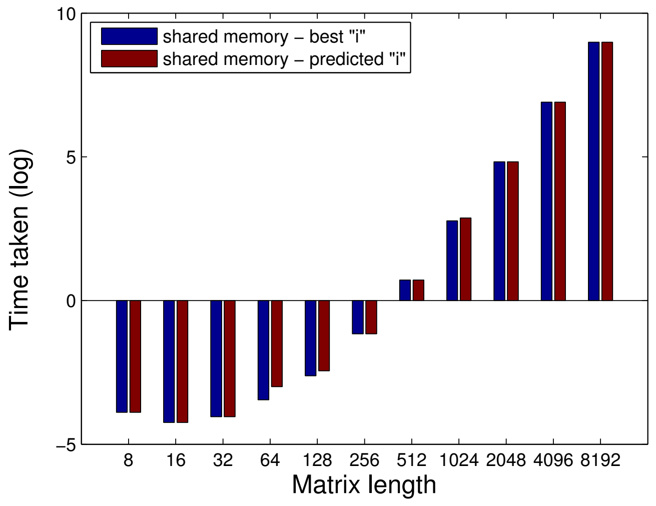


Figure 5-5: Comparison of results achieved- best i Vs predicted i. The graph spans both sides of the x-axis since the values on the y-axis are logarithm of the time.

optimal value of ‘i’ almost everywhere except at some instances(for matrix sizes 64, 128 and 1024).

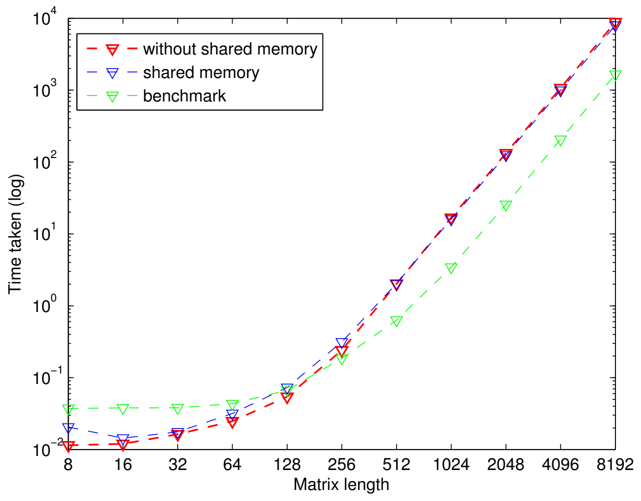


Figure 5-6: Performance Comparison of global memory and shared memory versions with CUBLAS benchmark

For comparison, we executed the SGEMM code of the cuBLAS library for different matrix sizes. The results have been plotted as a semilog graph and are shown in figure 5-6. Results show that our code (without shared memory) performs better for matrix sizes up to 128x128, but lags for greater matrix sizes. Thus, to get the best performance of the GPUs it is necessary to consider the cache hierarchy as well. Thus, taking into account such details and the need for complex atomic operations in real-life scenarios, it is necessary to effectively use the shared memory to enhance the performance and at the same time be simple enough to keep the productivity intact.

From this result, we see that cuBLAS provides an efficient implementation of matrix multiplication since it is a proprietary code and is heavily optimized. This led us to explore cuBLAS to see if we could integrate our approach onto cuBLAS. However, since we could not obtain the source code of cuBLAS from NVIDIA, we embarked in a different direction, by using cuBLAS as a black box for efficient computation on a single GPU, and using a framework comprising cuBLAS, powerlists and MPI to scale computation over a cluster of GPUs. We describe this method in detail in the next section.

5.4 Scaling Matrix Multiplication over a Cluster of GPUs

The NVIDIA CUDA [36, 35] Basic Linear Algebra Subroutines (cuBLAS) library [99] is a GPU-accelerated version of the complete standard BLAS library that delivers fast performance for computing the various subroutines(that involves matrix multiplication) on a single GPU. It consists of BLAS GPU routines that are optimized for dense matrices/vectors. However, the cuBLAS API does not support computation of these subroutines across a cluster of GPUs. Although a multi-GPU version of cuBLAS is available- known as cuBLAS-XT [99], it only supports scaling across multiple GPUs connected to the same motherboard. If we have several GPUs across a cluster, we cannot schedule an application across them to arrive at an energy efficient computation. Further, owing to resource constraints, it may not be possible for us to compute matrix multiplication of large matrices on a single GPU. For instance, Nvidia Tesla M2090 GPU has a memory of 6 GB, which is enough to multiply matrices with size less than 16384×16384. For larger matrices, the computation fails due to shortage of memory. This motivates us to explore the possibility of using appropriate scheduling to perform scaling up of the computation across a cluster of GPUs. To realize this, we propose a high level framework based on the abstraction provided by powerlists. We demonstrate our approach through the matrix multiplication problem for simplicity although this approach could also be applied to other problems like FFT, prefix-sum etc that are recursive in nature.

We utilize the powerlist data structure which facilitates automatic partitioning of the matrices and the cuBLAS API for efficient matrix multiplication of the sub-matrices at the individual GPUs. Powerlist allows us to represent several algorithms concisely because of the use of recursion and also because it avoids explicit indexing of a powerlist's elements using the operations defined on them. By employing these operations, it is possible to automate the partitioning process by eliminating the need for the user to partition the matrices. This abstraction provided by powerlist allows us to shield the user from the programming complexity resulting from the multi-level memory hierarchy. To achieve this, we have implemented powerlist operations as a library that automates the partitioning process. After partitioning, the appropriate sub-matrices are loaded to the GPUs as required, and then the SGEMM subroutine of the cuBLAS API is invoked to compute the matrix multiplication efficiently at the individual GPUs. The resultant sub-matrices are then gathered at the root site to obtain the resultant matrix. This method is similar to the map-reduce paradigm [102], where the matrices are mapped to appropriate partitioned matrices and sent to appropriate members of the clusters and the results are gathered to obtain the resultant matrix.

For performance evaluation, we compare the experimental results of our implementation with the matrix multiplication on a single GPU using cuBLAS. The results are very encouraging, and show that for matrices of size beyond 4096×4096, significant performance gain is achieved with our implementation on the cluster of GPUs.

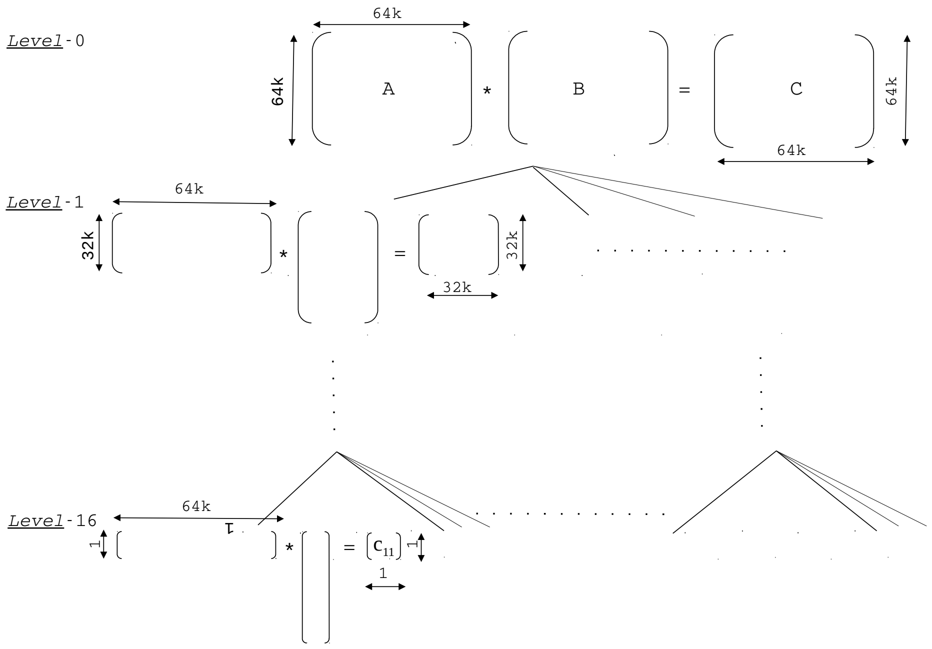


Figure 5-7: Tree representation of the matrix partitioning

5.4.1 Matrix Multiplication : An Illustration

Consider the computation of C = A * B, where all of A, B and C are matrices of size $64k \times 64k$. The complexity of the naive classical matrix multiplication algorithm is $O(n^3)$, where n is the length/width of the matrix. An efficient way to compute this in parallel is to use the blocked matrix multiplication algorithm as described in [36] that improves temporal locality. Later, in Section 5.2, we show how the blocked matrix multiplication [36] can be captured precisely using powerlists.

In our partitioning scheme, the matrix C is partitioned recursively into four smaller blocks of equal size, where A is partitioned horizontally into two, along the rows while B is partitioned along the columns. We demonstrate in Figure 5-7 how we can recursively partition the matrix multiplication computation into smaller sub-problems. At first level of recursion, A and B matrices are divided into two partitions, while matrix C is partitioned into 4. Thus, at level 1, the computation is divided into 4 sub-problems. At level 2, we have 16 subproblems. This goes on till level 16, when we get subproblems of size $[1 \times 64k]$*$[64k \times 1] = [1 \times 1]$.

Now, based on the number of GPUs in the underlying cluster(80 in our case) and the configuration of the individual GPUs, we might want to stop partitioning at some intermediate level so as to utilize the cuBLAS package to compute the individual subproblems efficiently, assuming that cuBLAS has been optimized with respect to the whole GPU. In general, at any level i, we get 4^i subproblems of size $[(64k/2^i) \times 64k]$*$[64k \times (64k/2^i)]$=$[(64k/2^i) \times (64k/2^i)]$, each of which are computed using cuBLAS at a single GPU.

The computation can be carried out in parallel at 4^i GPUs, each computing one of the 4^i blocks of matrix C, after loading the corresponding blocks of A and B matrices and utilizing the cuBLAS library to compute a block of matrix C. However, the disadvantage here is that the user is concerned with partitioning the matrices first, and then after computation, the resultant sub-matrices need to be stitched together appropriately to obtain the final result. This needs to be completely automated such that the user becomes oblivious to the partitioning concerns. Towards this, we describe a method as to how the powerlist notation can be used to express matrix multiplication, and how the partitioning and multiplication can be accomplished automatically using powerlists.

5.4.2 Computing matrix multiplication over GPUs

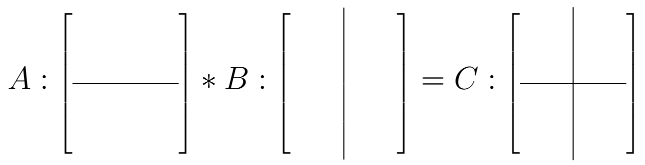
Since cuBLAS does not provide support for computation over a cluster of GPUs, it becomes difficult to compute large matrices efficiently by utilizing the capability of cuBLAS. To enable computation of larger matrices, we arrive at an algorithm to schedule the computation across a cluster of GPUs, using our representation with Powerlists while also utilizing the cuBLAS library for computing the subproblems on the individual GPUs.

Scheduling the computation across a cluster involves partitioning the matrices, which we propose to automate through Powerlists. Partitioning is achieved by using the tie and zip operations as deconstructors, that are defined on Powerlists. Similarly, the merging can be achieved by using the same tie and zip operations as constructors for combining the partial results. The advantage of using Powerlist is that it completely exposes the inherent concurrency in the application and allows us to have a concise functional description of parallel programs. Moreover, since index notations are not used, they allow us to work at a high level of abstraction.

Consider the computation of C = A * B. It can be specified in the Powerlist notation using rule 2.

Thus, at the first level of recursion, we get the following partitioning:

$$M(A_1|A_2, B_1|'B_2) \quad = \quad M(A_1, B_1)|'M(A_1, B_2)|M(A_2, B_1)|'M(A_2, B_2)$$



As we see, matrix A is partitioned into 2 submatrices by the *tie* operation. *tie* operation here is used as a *deconstructor* to partition A into 2 submatrices A_1 and A_2. Similarly B is also partitioned but vertically along the columns into B_1 and B_2 by applying *tie* in the second dimension i.e. along the columns.

This rule can be applied recursively for partitioning, where in at each *level* of recursion, A and B matrices are divided into two partitions, while matrix C is partitioned into 4. This corresponds to dividing the computation into 4 subproblems, and computing them using 4 GPUs. This shows how the level of recursion relates to the number of GPUs. The GPUs then individually compute the subproblems using the SGEMM subroutine of cuBLAS, following which the partial results are stitched together using the tie operation as *constructor* to obtain the resultant matrix, C.

5.4.3 Method of computing large matrices over GPU cluster using cuBLAS

In this section, we describe a method to automatically partition the matrices for efficient computation of matrix multiplication, schedule them on the GPU cluster and finally gather the partial results to obtain the resultant matrix.

Even though we have described the multiplication of matrices using Powerlist specifications in Section 5.4.2, the user is not required to give a Powerlist specification of the matrices. In practice, a matrix can be stored as a 1-D array in row-major layout[a] which can very well capture the representation of a matrix as a Powerlist.

Powerlists provide operations *zip* and *tie* that avoid explicit indexing of a Powerlist's elements. We utilize this property to automate the partitioning and merging of subproblems, and towards this, we implement these operations as a library, which allows us to treat these operations as a black box and use them.

The library is comprised of functions *tie* and *zip* as given below:

tie(A:Array(m,...n), dim:Integer, level:Integer, cons:Binary)

zip(A:Array(m,...n), dim:Integer, level:Integer, cons:Binary)

The parameters of the functions *tie* and *zip*, and their interpretations are described in Table5.7:

Parameter	**In/Out**	**Meaning**
Array(m,...,n)	in/out	array of size m*...*n
dim	input	the dimension on which operation is to be applied
level	input	level of recursion
cons	input	specifies whether operation is to be applied as a constructor(=1) or deconstructor(=0)

Table 5.7: description of parameters used in the library functions.

Both *tie* and *zip* functions take the array as input and perform the corresponding operation on the array, interpreting it as a Powerlist. The operation is applied on the dimension *dim*, at the level *l*, provided as input by the user. Each of the *zip* and *tie* operations can be used as both constructors and deconstructors, so although their implementations are different, we provide them as a single function, with an additional parameter-*cons* to differentiate them. For efficiency, we have implemented the functions for computation on a GPU.

For instance, the matrix A is represented as shown below:

[a]The actual implementation stores the matrices in column-major layout, as the cublassgemm subroutine for computation of matrix multiplication takes the input matrices in column-major layout.

$$A : \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix}$$

Then, the tie operation used for partitioning matrix A is represented as tie(A:Array(4,4), 1, 2, 0); This function partitions matrix A into 4 submatrices, along the columns, and stores them back into A.

We now explain the *MatrixMultiply* algorithm for computing matrix multiplication, as given in Algorithm 1. Consider matrices A and B, of dimensions $M \times N$ and $N \times P$ respectively. The *MatrixMultiply* procedure takes these matrices as input, along with the number of GPUs, Q, and gives the resultant matrix, C, where $C = A * B$.

It is possible to partition the original problem into a large number of subproblems. However, we may not be able to split the problem into larger number of logical subproblems. Hence, we may have to limit the level to which we recursively partition the matrices. Let us see how we arrive at an appropriate level, depending on the number of available GPUs, Q. From Figure 5-7, we see that at each level, the number of subproblems increases by a factor of 4. Thus, if we stop the recursion at level 1, we obtain 4 subproblems at the leaves of the tree, which corresponds to solving the 4 subproblems at 4 different GPUs, using cuBLAS. Similarly at level 2, we obtain $4^2 = 16$ subproblems, requiring 16 GPUs. Thus, $log_4 Q$ corresponds to the maximum level of recursion possible, based on the maximum possible subproblems that could be mapped to the available GPUs. Similarly, as the matrix A/B is partitioned horizontally/vertically into 2 at each level, the maximum possible level of recursion is $log_2 M/log_2 P$, respectively. We choose the minimum of these expressions as the default level(there is also an option for the user to specify the level), given by:

$$level = min(log_2(min(M, P), log_4 Q)); \tag{16}$$

Now, given this level, the procedure MatrixMultiply to compute the matrix multiplication can be given as in Algorithm 6.

Algorithm 6 *Algorithm*

```
 1: procedure MatrixMultiply($A(M,N), B(N,P), C(M,P), Q$)
 2:     $level = min(log_2(min(M,P), log_4 Q))$;
 3:     $numProcess = 4^{level}$;
 4:     tie(A(M,N), 0, level, 0); /* Partition A */
 5:     tie(B(N,P), 1, level, 0); /* Partition B */
 6:     for $i = 1$ to $numProcess$ do /*send partitions of A and B to the MPI processes */
 7:         $MPI\_Send(A + (j/2) * (Matrix\_Size/2), j)$
 8:         $MPI\_Send(B + (j\%2) * (Matrix\_Size/2), j)$
 9:     end for
10:     cudaMemCpy(d_A, A,..,cudaMemcpyHostToDevice);
    /* transfer A to the GPU device */
11:     cudaMemCpy(d_B, B,..,cudaMemcpyHostToDevice);
    /* transfer B to the GPU device */
12:     cublasSgemm(d_A, d_B, d_C,...);
    /* multiply matrices using cuBLAS */
13:     cudaMemCpy(C, d_C,..,cudaMemcpyDeviceToHost);
    /* transfer C to the host */
14:     MPI_Gather(C,..);
15:     for $i = 0$ to $4^{level}$ do /* Merge the partial results */
16:         $tie(C + i * \frac{M*P}{4^{level}}(\frac{M}{2^{level}}, \frac{N}{2^{level}}), 1, level, 0)$;
17:     $tie(C(\frac{M}{2^{level}}, N), 0, level, 1)$

end
```

The complete elucidation of the MatrixMultiply procedure is given below.

Elucidation of the MatrixMultiply Procedure

Consider the computation of C = A * B, where

$$A : \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} * B : \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = C : \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

Suppose the target cluster has 4 GPUs. From step 2, we get level = 1. This is because although the maximum level of recursion based on the dimensions(M = P = 4) is 2, at the second level of recursion, total 16 GPUs would have been required. So for level 1, we get numProcess = 4 from step3. Steps 4 and 5 correspond to partitioning of A and B matrices. It is given by recursion on *tie*(dim=0) to A, and on *tie*(dim=1) to B at first level of recursion. This gives us the partitions of A and B at level one as follows:

$$A0 : \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \end{bmatrix} A1 : \begin{bmatrix} a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

$$B0 : \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{bmatrix} B1 : \begin{bmatrix} b_{1,3} & b_{1,4} \\ b_{2,3} & b_{2,4} \\ b_{3,3} & b_{3,4} \\ b_{4,3} & b_{4,4} \end{bmatrix}$$

Now, given these partitions, the appropriate submatrices are to be sent by the root process to the other processes. Since we use MPI(Message Passing Interface) for communication over the cluster, this is accomplished using the MPI_send procedure as given in steps 6-9. These steps are carried out by the root process, while the non-root processes invokes the MPI_Recv procedure to receive the input submatrices.

In steps 10-11, each of the processes copy the partitioned submatrices of A and B to their GPU devices using cudaMemCpy. At this stage, all the input submatrices are available at the GPUs. In step 12, the *cublasSgemm* subroutine of the cuBLAS library is invoked by the processes to compute the partial results on the respective GPUs. As a result, we obtain the 4 submatrices C0, C1, C2 and C3 of matrix C, where,

$$C0 : \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix} C1 : \begin{bmatrix} c_{1,3} & c_{1,4} \\ c_{2,3} & c_{2,4} \end{bmatrix} C2 : \begin{bmatrix} c_{3,1} & c_{3,2} \\ c_{4,1} & c_{4,2} \end{bmatrix} C3 : \begin{bmatrix} c_{3,3} & c_{3,4} \\ c_{4,3} & c_{4,4} \end{bmatrix}$$

Step 12 offloads the data from the GPU device, back to the host process. In step 13, the partial results at all the processes are gathered at the root process using MPI_Gather. At this stage, all the submatrices gets stored into the matrix, C. However, these submatrices need to be stitched together appropriately so as to obtain the final resultant matrix, C. This is accomplished using steps 14-16 by the root process. Step 15 is performed twice, once to combine C0 and C1, and then to combine C2 and C3 along the columns. This gives us the following submatrices:

$$C01 : \begin{bmatrix} c1,1 & c1,2 & c1,3 & c1,4 \\ c2,1 & c2,2 & c2,3 & c2,4 \end{bmatrix} C23 : \begin{bmatrix} c3,1 & c3,2 & c3,3 & c3,4 \\ c4,1 & c4,2 & c4,3 & c4,4 \end{bmatrix}$$

This is followed by step 16, which is trivial since it just needs to concatenate the submatrices along the rows, thereby providing the final resultant matrix, C.

Generality of the specifications

Powerlists require its dimensions to be a power of two. For matrices with dimensions not a power of 2, the computation can still be ensured by padding the matrices appropriately. Padding can be an important performance concern. In this section, we show how padding is accomplished and that it does not impede the performance significantly. Consider a matrix of size 36x32. This could be padded up by zeros to a matrix of size 64x64. Till now, we have considered only square matrices of dimensions $2^n \times 2^n$ for illustration. However, our approach applies to rectangular matrices as well without any additional efforts. So now, the matrix needs to be padded up to a matrix of dimensions 64x32. In general, any matrix with dimensions $(2^m + l) \times 2^n$, where $l < 2^m$ needs to be padded up to a matrix of dimensions $2^{m+1} \times 2^n$.
Experiments to study the effect of this padding revealed significant overhead, with up to 90% slowdown in some computations.

A solution to this is to pad the matrix with dimensions $(2^m + l) \times 2^n$ to another

matrix with dimensions $(2^m + 2^t) \times 2^n$, where, $2^{t-1} < l \leq 2^t$. So now, the computation of $C = A * B$, where A is a matrix of size $(2^m + l) \times 2^n$, B is of size $(2^n) \times 2^p$, is given by $C = A1 * B | A2 * B$,
where $A1$ is of size $2^m \times 2^n$, and $A2$ of $2^t \times 2^n$.

To illustrate this, let us consider the computation $A * B = C$, where A is of size 70×64, B of size 64×64, C of size 65×64. Instead of padding A to a matrix of size 128×64, we pad it by zeros to the matrix A' of size 72×64. Now, the computation $C = A' * B$ can be split into two as $C = A1 * B | A2 * B$, where $A1$ is of size 64×64 and $A2$ of size 8×64. This involves considering them as two independent problems and computing them using our method. The resultant matrix C is then obtained by applying tie operation on the two results which is as simple as concatenating the two resultant submatrices.

With this solution, the overhead now can reduced considerably, thereby allowing us to generalize our method for matrices whose dimensions are not a power of two, without affecting the performance significantly. We are also working on extending this method to ParLists[105](that is an extension of Powerlist notation to lists of arbitrary positive lengths), which is able to naturally capture this generalization.

5.4.4 Experimental Results

The experimental results for the matrix multiplication computation over the cluster show significant performance gains for large dimensional matrices. For matrix sizes greater than 2048x2048, relatively better performance is achieved over the cluster as compared to the computation on a single GPU, with upto 132% gain achieved for matrices of size 16384x16384. From the experimental results, it appears that as we further increase the matrix size, the gain would be even higher. First, we describe the configuration of the GPU cluster.

Configuration of the cluster:
The cluster is comprised of 40 nodes, where each node consists of

1. 2 Xeon servers, each having 6 processors, total = 12 processors per node.

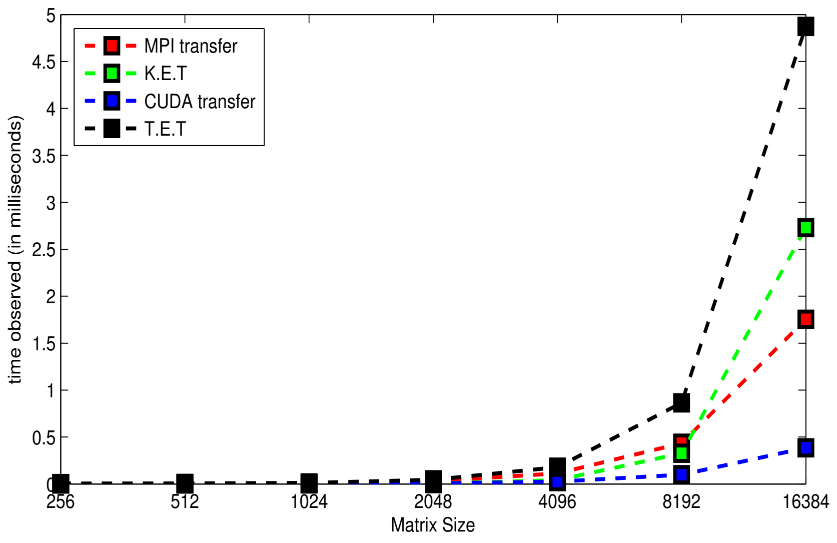


Figure 5-8: Execution with 4 processes, combining all the MPI transfers together, and CUDA transfers

2. 2 Tesla M2090 GPUs, each having 512 cores, Processor core clock: 1.3 GHz, 6GB GPU memory.

Thus in all, the cluster consists of 80 GPUs.

The computation of matrix multiplication involves n processes run on n CPUs, with the 1st one called root process. The computation involves communication between various nodes, which we accomplish using MPI. The steps for the entire computation have been listed in the algorithm discussed in section 5.4.3.

We call the total time taken for computing the procedure as the 'Total Execution Time(TET)' and time taken in step 12 as the 'Kernel Execution Time(KET)'. We call the time taken in steps 6-9 and 14 together as MPI Transfer time, and that for steps 10, 11 and 13 as CUDA memcpy time. We observe the TET and KET for matrices of sizes from 256x256 upto 32768x32768, and for each matrix size, we repeat the experiment by varying n=1, 4, 16 and 64.

Observations:

The kernel execution time increases by a factor of 8 as the matrix size increases for

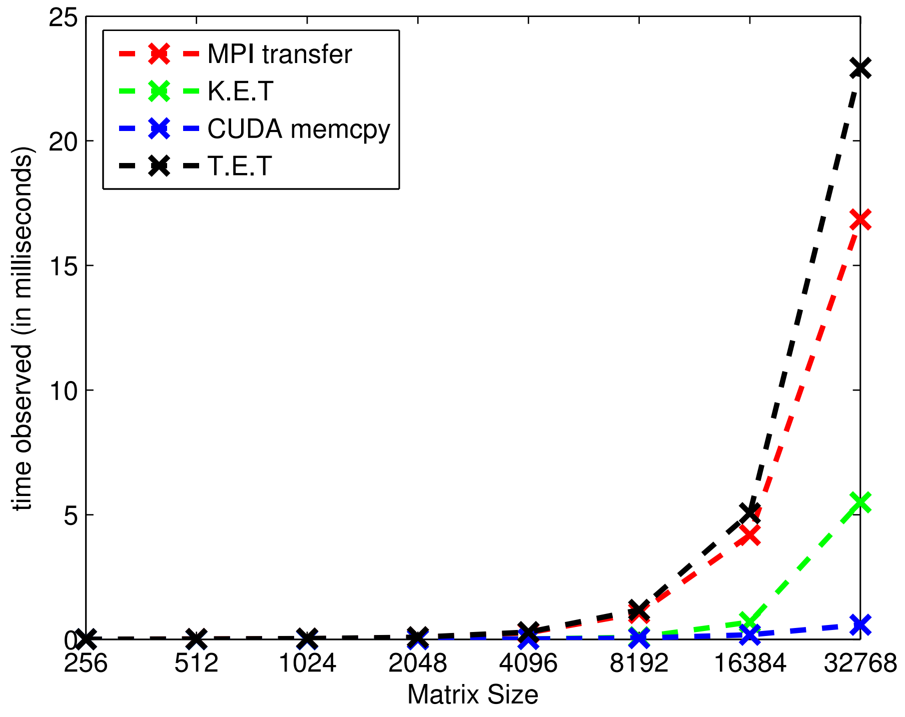


Figure 5-9: Execution with 16 processes, combining all the MPI transfers together, and CUDA transfers

the same number of processes, and it decreases by a factor of 4 as the number of processes increases by 4 for the same matrix size, as expected.

From Figures 2-d and 2-e, we observe that the best performance is achieved on a single GPU for smaller matrix sizes till 2048x2048. This can be explained form the fact that the MPI transfer time dominates the kernel execution time for multiple GPUs, as is evident from Figures 2-a, 2-b and 2-c, whereas for the single GPU, the communication overhead of MPI transfers is absent. However for matrix sizes above 2048x2048, better performance is achieved on computing it across 4 GPUs as compared to computing it on a single GPU. Although the performance gain is just 13.07% for matrix of size 4096x4096, it increases significantly for larger matrix sizes, with a 132% gain achieved for matrices of size 16384x16384. Another point to be noted from Figures 2-d and 2-e, the graph for T.E.T for 4 processes approaches the graph for 16 processes as matrix size increases, and is in fact approximately equal for matrix size 16384x16384, indicating that the performance for 16 processes may be

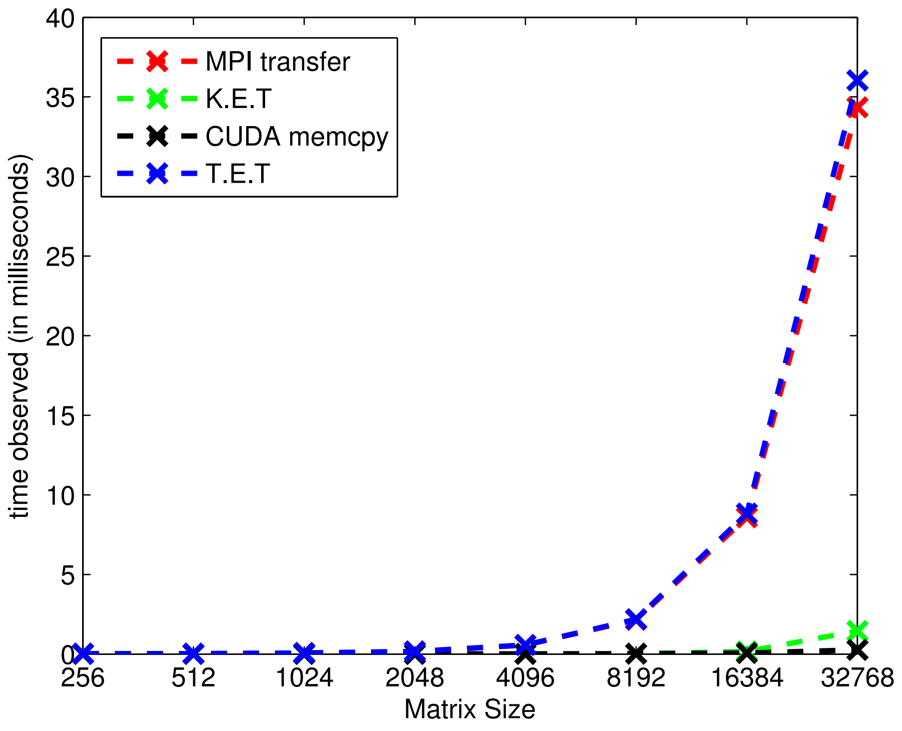


Figure 5-10: Execution with 64 processes, combining all the MPI transfers together, and CUDA transfers

even better for higher matrix sizes. The summary of these details of the number of GPUs for which the best performance is achieved, has been listed in Table 5.8.

Matrix Size	**Number of GPUs**	**Level**
upto 2048x2048	1	0
4096x4096 to 16384x16384	4	1
above 16384	16	2

Table 5.8: shows the number of GPUs for which the best performance is achieved for different matrix sizes.

Analysis of the algorithm and arriving at the optimal recursion level

The experiments show the superiority of our approach in running large matrices that would not have been possible otherwise. For example, the GPUs used in our experiments are able to handle matrices of size only upto 16384x16384. Whereas our

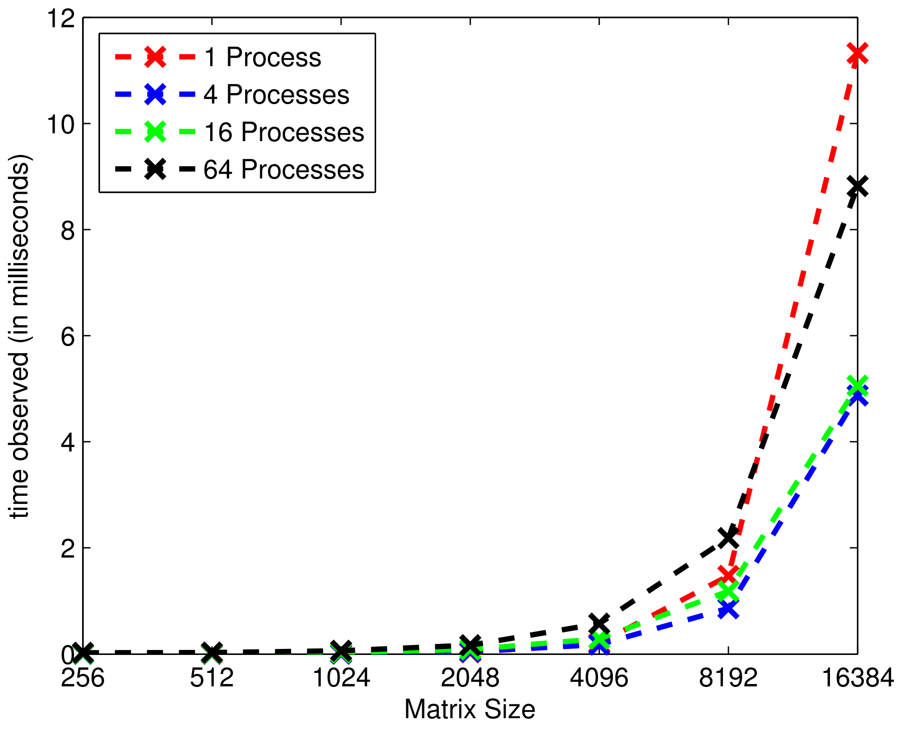


Figure 5-11: Comparison of Total Execution times for 1, 4, 16 and 64 processes

algorithm computes matrices of size more than 16384x16384 as well across multiple GPUs.

With the above approach, the computation can be scheduled across a cluster of GPUs. However, the user still needs to manually tune the application for optimum performance by changing the level of recursion which in turn decides the number of GPUs across which the computation spans. One way to automate tuning is to use Empirical Search as discussed in [38],[39] and [40]. In this section, we show an analytical approach to arrive at the level of recursion at which the best performance can be achieved.

Let us consider the matrix multiplication of matrices A of size $M \times N$, and B of size $N \times P$ to matrix C of size $M \times P$. Then, let $T(M, N, P)$ be the KET to compute C on a single GPU, using cuBLAS. Since the computation is recursive in nature, we can give the recurrence relation on T by,

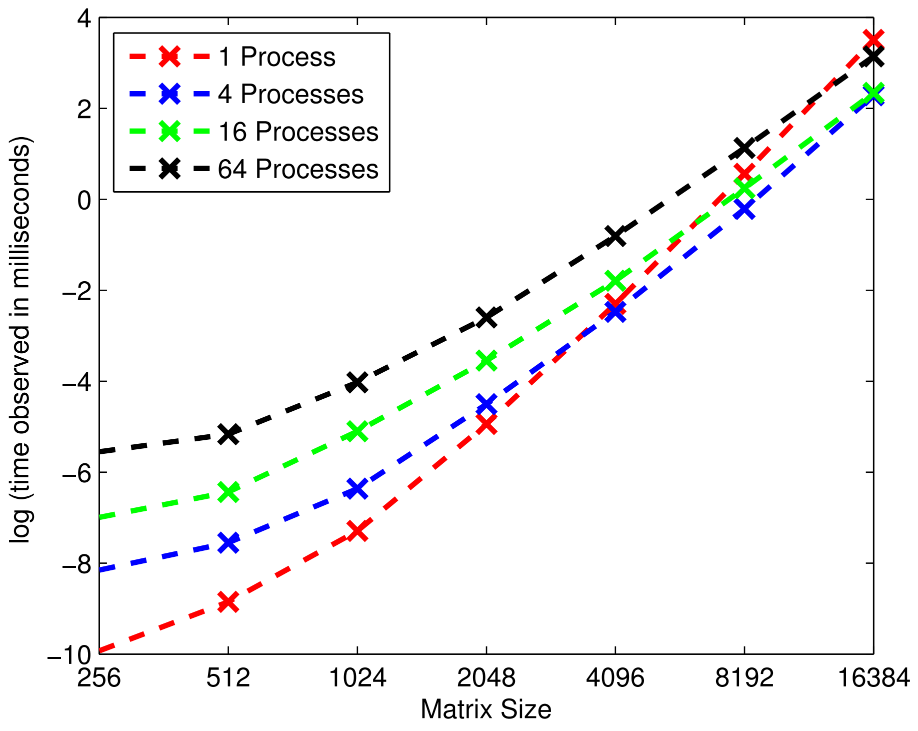


Figure 5-12: Comparison of Total Execution times for 1, 4, 16, 64 processes(in log)

$$T(M, N, P) = 2^{2i} * T(M/2^i, N, P/2^i) + C_1 \tag{17}$$

where, C_1 is a constant which denotes the overhead incurred in partitioning and merging. This shows that the KET for computing C would be smallest when $2^{2i} * T(M/2^i, N, P/2^i)$ is smallest. However, since the computation using cuBLAS spans the entire GPU, we need to partition the matrix into the largest possible submatrices that can fit into the GPU's memory, so as to get the best performance. This is also true because further partitioning would incur additional overheads of merging the resultant submatrices. Thus, matrices of size up to 16384x16384 would not entail any further partitioning. Now, let us consider a matrix of size 32768x32768. At level 1, 4 subproblems are obtained where A is of size 16384×32768, B of size 32768×16384, and C of size 16384×16384. Since the matrices cannot be handled by a single GPU, they need to be partitioned further. We could divide matrix A horizontally into A1 and A2 and matrix B vertically into submatrices B1 and B2, such that C matrix is given by,

$C = (A1 * B1|'A1 * B2)|(A2 * B1|'A2 * B2)$

The computation would now involve computing submatrices $A1*B1, A1*B2, A2*B1$ and $A2*B2$ sequentially and then merging them to obtain C. In general, this involves partitioning the matrices recursively until each of the partitioned submatrices are able to fit in the GPU memory, computing the matrix multiplications of the corresponding submatrices in serial, and then merging them. This also shows how our approach could be scaled for computation of matrices of any arbitrary size that are limited by the GPU's memory but not by the host's memory.

Now, let T(M, N, P) be the KET to compute C across multiple GPUs, say N GPUs, using cuBLAS. The recurrence relation on T can now be given by,

$$T(M, N, P) = \lceil 2^{2i}/N \rceil * T(M/2^i, N, P/2^i) \tag{18}$$

From our experiments, we find that $T(M/2^i, N, P/2^i) \simeq T(M, N, P)/2^{2i}$, which is also consistent with equation 17. Thus, we get

$$T(M, N, P) = \lceil 2^{2i}/N \rceil / 2^{2i} \tag{19}$$

Now, the optimal value of i can be found by considering this expression of time as a function of i and minimizing this function for the values of i.

However, in order to get the optimal depth of recursion, we might also have to consider the MPI data transfer time incurred in the computation. Figures 2-a, 2-b and 2-c show that the MPI transfer time increases rapidly with increase in matrix size and with the number of processes, and hence plays a significant role in the performance. We are working towards trying to capture the MPI transfer time as a function of i that would allow us to analytically arrive at the optimal level of recursion. We are also working towards reducing the time incurred in MPI transfer by using pipelining to parallelize the computation and the data transfer.

Scaling the computation

A GPU may not be able to handle matrices beyond a certain size due to memory limitations. Even though we partition the matrices across a cluster, the partitioned matrices may still be too large for the GPU memory. In such a scenario, we could further partition the matrices at the GPU level. For example consider the example from section 2, wherein, at level 1 of recursion, the following partition is obtained and is to be computed:

$$A1 : \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \end{bmatrix} * B1 : \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{bmatrix} = C1 : \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}$$

Now suppose the GPU memory is not large enough to hold both A1 and B1 matrices simultaneously. We could further divide A1 matrix horizontally into A11 and A12 and B1 matrix vertically into submatrices B11 and B12, such that C1 matrix is given by,
$C1 = (A11 * B11|'A11 * B12)|(A12 * B11|'A12 * B12)$

$$A1 : \left[\begin{array}{cccc} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \hline a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \end{array}\right] * B1 : \left[\begin{array}{c|c} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{array}\right] = C1 : \left[\begin{array}{c|c} c_{1,1} & c_{1,2} \\ \hline c_{2,1} & c_{2,2} \end{array}\right]$$

The computation would now involve computing submatrices $A11 * B11, A11 * B12, A12 * B11$ and $A12 * B12$ sequentially and then merging them to obtain C1. This would involve partitioning the submatrices recursively until each of the partitioned submatrices are able to fit in the GPU memory, computing the matrix multiplications of the corresponding submatrices in serial, and then merging them. This is similar to the partitioning and merging at the cluster level, except that here the computation is performed in serial.

Scaling FFT

The Discrete Fourier Transform is an important tool used in many scientific applications, especially in digital signal processing. It maps a sample from a cycle of data points of a periodic signal onto a frequency spectrum representation containing the same number of points. It is mainly used for time series analysis, convolutions and to solve partial differential equations.

Given

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

its Discrete Fourier Transform is given by the vector

$$y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where, each $y_k = \sum_{j=0}^{n-1} e^{-2\pi kji/n} x_j$ for k = 0, 1, ...,n-1.

The Fast Fourier Transform, or FFT, is a method to compute the Discrete Fourier Transform popularized by Cooley & Tukey. It was developed as a "quick" method for computing the Discrete Fourier Transform. Various methods of FFTs were implemented in a wide variety of applications. There are many types of FFT algorithms, but they fall into one of the two general categories: decimation in time approach and decimation in frequency approach. The decimation in time approach breaks apart the time dependent input vector x in order to approximate the solution. The decimation in frequency approach breaks apart the frequency dependent vector y in its computation.

Misra derived FFT algorithm from the Cooley-Tukey algorithm [100, 101], and is given as below:

$$fft. < a >=< a > \tag{20}$$

$$fft.(pq) = fft.p + u * fft.q | fft.p - u * fft.q \tag{21}$$

where, $u =< \omega^0, \omega^1, ..., \omega^{N-1} >$, N is the length of p, ω is the $2 * N$th principal root of 1.

Radix-R Stockham algorithm is a FFT algorithm that avoids the index-shuffling stage using Stockham formulations of the FFT. We now discuss a GPU implementation of the radix-R Stockham algorithm.The algorithm is given in Figure 5-13. The number of threads used for GPU_FFT(), T, is N/R. In each iteration over the data, R subarray of length Ns are combined into arrays of length RNs. The iterations stop when the entire array of length N is obtained. The data is read from memory and scaled by the twiddle factors, combined using an R-point FFT, and written back out to the memory. The expand() function inserts a dimension of length N_2 after the first dimension of length N_1 in a linearized index.

CUDA provides support for FFT computation in the form of cuFFT library. The cuFFT library is a highly optimized and tested FFT library that provides a simple interface to the user for computing FFTs on an NVIDIA GPU, allowing them to leverage the floating-point power and parallelism of the GPU. We utilize the subroutine cufftExecC2C provided by cuFFT to compute FFT.

To show the efficiency of cuFFT, we compare the execution times of cufftExecC2C with that of the radix-R Stockham implementation given in Figure 5-13, on same inputs. Table 5.9 shows the performance of the radix-R Stockham implementation. The experiment involved varying the level of recursion to vary the number of blocks and the input size. The results demonstrate significant performance gain achieved by dividing the computation into smaller subproblems which is upto 2x for larger input sizes, however, the increase in execution time is not proportional to the increase in input size.

Table 5.10 shows the performance of cuFFT.

```
float2* CPU_FFT(int N, int R,
                float2* data0, float2* data1) {
  for( int Ns=1; Ns<N; Ns*=R ) {
    for( int j=0; j<N/R; j++ )
      FftIteration( j, N, R, Ns, data0, data1 );
    swap( data0, data1 );
  }
  return data0;
}

void GPU_FFT(int N, int R, int Ns,
             float2* dataI, float2* dataO) {
  int j = b*N + t;
  FftIteration( j, N, R, Ns, dataI, dataO );
}

void FftIteration(int j, int N, int R, int Ns,
                  float2* data0, float2*data1){
  float2 v[R];
  int idxS = j;
  float angle = -2*M_PI*(j%Ns)/(Ns*R);
  for( int r=0; r<R; r++ ) {
    v[r] = data0[idxS+r*N/R];
    v[r] *= (cos(r*angle), sin(r*angle));
  }
  FFT<R>( v );
  int idxD = expand(j,Ns,R);
  for( int r=0; r<R; r++ )
    data1[idxD+r*Ns] = v[r];
}

void FFT<2>( float2* v ) {
  float2 v0 = v[0];
  v[0] = v0 + v[1];
  v[1] = v0 - v[1];
}

int expand(int idxL, int N1, int N2 ){
  return (idxL/N1)*N1*N2 + (idxL%N1);
}
```

Figure 5-13: GPU implementation of radix-R Stockham algorithm [104]

Table 5.9: Execution times of radix-R Stockham implementation

level	Blocks	Input Size					
		8	32	128	512	2048	8192
0	1	158	529	2001	9468	45128	211680
1	2	113	356	1133	5129	24138	113009
2	4	101	251	704	2948	13610	63459
3	8		211	490	1872	8425	39181
4	16		210	323	1319	5798	27024
5	32			245	750	3030	13646
6	64			211	427	1659	7041
7	128				309	942	3741
8	256				284	554	2202
9	512					429	1408
10	1024					392	884
11	2048						657
12	4096						616

Table 5.10: Execution times of cuFFT

Input Size					
8	32	128	512	2048	8192
48	58	70	73	75	87

It shows that cuFFT is upto 7 times faster than the radix-R Stockham implementation. Moreover, cuFFT internally identifies the best kernel execution parameters for the inputs, and hence relieves the user from the task of identifying them. To leverage the performance of cuFFT, we utilize cuFFT to scale FFT computation across a cluster of GPUs.

Our approach

We utilize Powerlists and the cuFFT library to scale FFT computation across a cluster of GPUs. As before, we utilize powerlists to automatically partition the computation into subproblems(at a certain level of recursion), in the form of the library implementation of powerlist operations zip and tie. We then compute the subproblems independently at the various GPUs using the cuFFT library. This is followed by merging the results at each level using the Powerlist library, till we obtain the final result. The procedure to compute FFT could be listed as follows:

Procedure FFT

1. Partition the input to level m using Powerlist operations as deconstructor.
2. Transfer the input data to 2^m GPUs using MPI_Send.
3. Compute the FFT using cufftExecC2C subroutine of the cuFFT library for each of the 2^m subproblems.
4. Collect the partial results using MPI_gather.
5. for $level = m \to 1$

   Merge the corresponding subproblems using the Powerlist operations as constructor.

Thus, we have shown how our approach can be extended to scale FFT on cluster of GPUs using powerlists to partition the problems, and merge the partial results, cuFFT libbrary to compute the subproblems at the individual GPUs, and MPI to realize communication between different nodes. This approach can be adapted similarly for other applications as well, that are recursive in nature.

We now describe our observation about the relationship between powerlists and oblivious algorithms in the next section.

5.5 Powerlists and Oblivious algorithms

Several applications can be easily formulated with the divide and conquer strategy, which are solved using recursion. And since recursive algorithms exhibit good temporal locality, they execute efficiently on computers with caches. Cache-aware algorithms [120] make use of the cache parameters, and let the programmers tune them for performance. Cache-oblivious algorithms are oblivious to the cache parameters, and consequently are simpler and more portable as compared to cache aware algorithms. Multicore-Obivious (MO) [121] algorithms are oblivious of the multicore parameters in addition to the cache parameters. Unlike cache oblivious algorithms that are sequential algorithms, MO algorithms utilize the parallelism offered by the multiple cores, and hence, are more performance efficient. In this work, we report our observation that most of Cache Oblivious and Multicore Oblivious algorithms are recursive in nature, and how they can be easily expressed using powerlists, a notation for describing parallel programs.

5.5.1 Cache-Oblivious Algorithms

A cache-oblivious algorithm [120] makes no use of any cache parameters like the number of cache levels, cache sizes and block lengths. They are designed for efficient usage of cache memories and the available memory bandwidth. A cache aware algorithm, on the other hand, requires certain parameters to be tuned for optimal cache complexity. The programmer then is responsible for manually tuning this parameter to achieve optimal performance. For example, value of the particular parameter may need to be chosen such that the subproblems simultaneously fit into the cache. Since, the oblivious algorithms are devoid of such parameters, they offer advantages of simplicity and portability. In the next section, we show how Powerlists can be used to succinctly express Cache Oblivious algorithms.

Expressing Cache-Oblivious Algorithms using powerlists

We observed that all cache-oblivious algorithms are recursive in nature, and that they can be easily specified using Powerlists. We now show the powerlist specifications for the various cache-oblivious algorithms proposed in literature.

Matrix Multiplication

Given a mxn matrix A and a nxp matrix, B, we first show how the cache oblivious algorithm for multiplying A and B, to give a mxp matrix C. The matrices are assumed to be stored in row-major layout. The algorithm is based on divide-and-conquer. It divides the largest partition into half, and recurs according to the following cases:

The base case for this recursion is given by $m = n = p = 1$, which is straight forward. When $m \geq$max(n, p), Matrix A is split horizontally into A1 and A2, and

$$\begin{aligned}
AB &= \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \\
AB &= \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1B_1 + A_2B_2 , \\
AB &= A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} .
\end{aligned}$$

both are then multiplied by B, and likewise for $p \geq$maxm, n. However, when $n \geq$ max(m, p), A is partitioned horizontally and B vertically, and the result is given by $A_1B_1 + A_2B_2$. The algorithm works by recurring as per the above cases until a subproblem fits into the cache. Now, the smaller subproblems can be solved without any more cache misses.

Powerlist Specification for Cache-Oblivious Matrix Multiplication

We now express the cache oblivious matrix multiplication by giving equivalent powerlist specifications for the three cases discussed above.
Base Case: Matrix A and B are 1×1 matrix, i.e., $A = \langle\langle x \rangle\rangle$, and $B = \langle\langle y \rangle\rangle$. Then,

$$M(\langle\langle x \rangle\rangle, \langle\langle y \rangle\rangle) = \langle\langle x \times y \rangle\rangle \tag{22}$$

Here, $\langle\langle x \rangle\rangle$ and $\langle\langle y \rangle\rangle$ are singleton elements (or a 1×1) matrix and $\times$ denotes the classical multiplication of numbers.

Case 1: When $m \geq$maxn, p, the equivalent powerlist specification is given below: Let $A = A_1|A_2$; Then

$$M(A_1 \mid A_2, B) = (M(A_1, B) \mid M(A_2, B)) \tag{23}$$

Case 2: The equivalent powerlist specification for the case when $p \geq$maxm, n is given by Let $B = B_1|'B_2$; Then

$$M(A, B_1 \mid' B_2) = (M(A, B_1) \mid' M(A, B_2)) \tag{24}$$

Case 3: When $n \geq \max m, p$, Let $A = A_1|A_2$, and $B = B_1|B_2$; Then

$$M(A_1 \mid A_2, B_1 \mid' B_2) = (M(A_1, B_1) \mid' M(A_1, B_2)) \mid (M(A_2, B_1) \mid' M(A_2, B_2)) \quad (25)$$

Thus, the 3 rules described here correspond to the the 3 rules defined in the earlier section, thereby illustrating the capability of powerlists to specify cache oblivious matrix multiplication. We now give brief overview of Multi-core Oblivious algorithms, and show how the powerlists can be used to specify Multi-core Oblivious algorithms through Matrix Transposition and Fast Fourier Transform (FFT).

5.5.2 Multi-core Oblivious Algorithms

Cache oblivious algorithms are sequential algorithms that do not make use of cache parameters, thereby offering advantages of simplicity and portability. Multicore oblivious algorithms (MO)[121], in addition, attempt to exploit the parallelism offered by multicore architectures. MO algorithms are oblivious of multicore and cache parameters, but are allowed to provide hints to the runtime scheduler that is aware of the multicore parameters. These hints facilitate scheduling of the algorithm on the multicore architectures with a multilevel cache hierarchy and multiple cores. A hierarchical multi-level caching model has been presented in [121] that consists of h levels of cache and p cores. The cores share an arbitrarily large shared memory through the cache hierarchy.

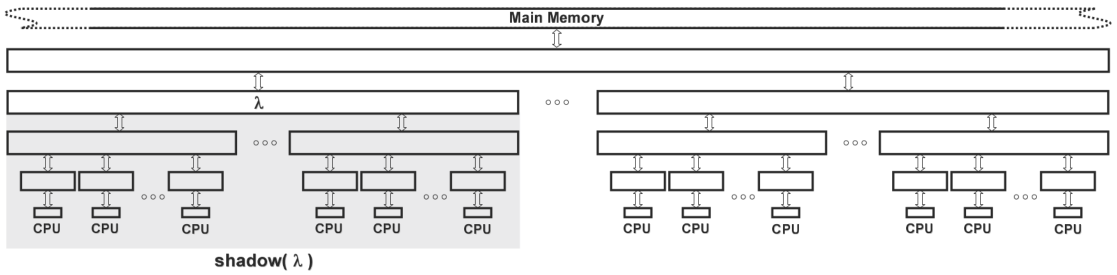


Figure 5-14: The HM model [121] for 5 levels of cache

Figure 5-14 illustrates the HM model for h=5. The type of scheduling hints employed by MO algorithms are:

1. coarse-grained contiguous (CGC): distributes parallel fine-grained subtasks in contiguous chunks across a sequence of contiguous cores. This hint is used to decompose a parallel pfor loop into segments executed in parallel by contiguous cores.

2. space-bound (SB): an upper bound on the space used by each forked task. This hint is applied for algorithms that recursively spawn parallel tasks.

3. CGC on SB (CGC $\Rightarrow$ sb): combines both CGC and SB. This hint is useful for algorithms that recursively spawn parallel tasks.

We now show how powerlists can be used to specify MO algorithms. In particular, we show how the multicore-oblivious Fast Fourier Transform (MO-FFT), as presented in [121], can be described using powerlists. However, since MO-FFT internally uses matrix transposition, we first arrive at a multicore oblivious matrix transposition algorithm (MO-MT), and specify it using powerlists. This powerlists specification of MO-MT is then used to describe MO-FFT.

Matrix Transposition

A multicore-oblivious matrix transposition algorithm (MO-MT) has been presented in [121] that uses CGC hints. The algorithm is shown in Figure 5-15.

```
MO-MT($A$, $n$)
Input: an $n \times n$ matrix $A$ in row-major order.
Output: the transpose $A^T$ in row-major order.
1: [CGC] pfor $0 \le z \le n^2 - 1$ do
2:         $(i, j) = \beta(z)$;
3:         $A^T[j, i] := A[i, j]$;
```

Figure 5-15: multicore-oblivious matrix transposition algorithm

The algorithm scans the elements of matrix A according to the bit-interleaved layout and maps them to their respective positions in the transformed matrix A^T. However, since this algorithm involves index mapping, we present a recursive definition of Matrix Transposition (MO-MT_Rec) that is easier to specify using powerlists, and uses SB hints. Since matrices have 2 dimensions, we use powerlists of powerlists to express matrices. For eg. $<< a0\ a1 > < a2\ a3 >>$ represents a 2x2 matrix, where the matrix $< a0\ a1 >$ represents the first row and $< a2\ a3 >$ represents the second row. MO-MT_Rec can then be defined as:

$$\begin{gathered} T << x >>=<< x >> \\ T((p|'q)|(r|'s)) = (T(p)|'T(r))|((T(q))|'(T(s))) \end{gathered} \tag{26}$$

We now give describe the multicore oblivious FFT algorithm MO-FFT in the next section.

Multicore Oblivious Fast Fourier Transform

The Discrete Fourier Transform is an important tool used in many scientific applications, especially in digital signal processing. It maps a sample from a cycle of data points of a periodic signal onto a frequency spectrum representation containing the same number of points. It is mainly used for time series analysis, convolutions and to solve partial differential equations.

Given $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$ its Discrete Fourier Transform is given by the vector $y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$

where, each $y_k = \sum_{j=0}^{n-1} e^{-2\pi kji/n} x_j$ for k = 0, 1, ...,n-1.

The Fast Fourier Transform, or FFT, is a method to compute the Discrete Fourier Transform popularized by Cooley & Tukey. It was developed as a "quick" method for computing the Discrete Fourier Transform. Various methods of FFT's were implemented in a wide variety of applications. Misra derived FFT algorithm from the Cooley-Tukey algorithm, and is given as below:

$$fft. < a > = < a >$$
$$fft.(p \bowtie q) = (fft(p) + u * fft(q)) | (fft(p) - u * fft(q)) \tag{27}$$

where, $u = < \omega^0, \omega^1, ..., \omega^{N-1} >$, N is the length of p, ω is the $2 * N$th principal root of 1.

Since our main objective is to specify multicore oblivious algorithms using powerlist, we first present the MO-FFT algorithm from [121], and then arrive at the powerlist specification for this algorithm. The MO-FFT algorithm views the input as a square matrix, which it transposes, then performs a sequence of two recursive FFT computations on independent parallel subproblems, and finally performs Matrix transpose on the result. The algorithm is given in Figure 5-16

```
MO-FFT($X, n$)
**Input:** A vector $X$ of length $n = 2^k$ for some integer $k \geq 0$.
**Output:** In-place FFT of $X$.
**Space Bound:** $S(n) = 3n$.
 1: **if** $n$ is a small constant **then** compute FFT using the direct formula and **return**.
 2: Let $n_1 = 2^{\lceil \frac{k}{2} \rceil}$ and $n_2 = 2^{\lfloor \frac{k}{2} \rfloor}$ (observe that $n_1 \in \{n_2, 2n_2\}$), and let $A$ be an $n_1 \times n_2$
    matrix stored in row-major order.
 3: **[CGC] pfor** $0 \leq i < n_1, 0 \leq j < n_2$ **do** $A[i,j] := X[i \cdot n_2 + j]$
 4: **[CGC]** MO-MT($A, n_1$). (When $n_1 = 2n_2$, the matrix is computed by splitting the
    matrix into two square matrices and then invoking MO-MT twice.)
 5: **[CGC⇒SB] pfor** $0 \leq i < n_2$ **do** MO-FFT($A[i, 0 \ldots (n_1 - 1)], n_1$)
 6: **[CGC]** Multiply the $n$ entries of $A$ by appropriate twiddle factors
 7: **[CGC]** MO-MT($A, n_1$)
 8: **[CGC⇒SB] pfor** $0 \leq i < n_1$ **do** MO-FFT($A[i, 0 \ldots (n_2 - 1)], n_2$)
 9: **[CGC]** MO-MT($A, n_1$)
10: **[CGC]** Copy the $n$ entries of $A$ into $X$
```

Figure 5-16: The multicore oblivious FFT algorithm [121]

MO-FFT is essentially a 6-step algorithm, the first 3 steps and the last step of MO-FFT are trivial. Thus MO-FFT boils down to the following 6 steps:

1. MO-MT(A, n_1)

2. pfor $0 \leq i < n2$ do MO-FFT(A[i,o...(n_1-1)], n_1)

3. Multiply the n entries of A by appropriate twiddle factors

4. MO-MT(A, n_1)

5. pfor $0 \leq i < n1$ do MO-FFT(A[i,o...(n_1-1)], n_2)

6. MO-MT(A, n_1)

Steps 1, 4, and 6 compute matrix transposition, which has already been discussed and can be specified as:

$$T << x >>=<< x >>$$
$$T((p|'q)|(r|'s)) = (T(p)|'T(r))|((T(q))|'(T(s)))$$

Step 3 can be specified using the scalar operation * which multiplies each element of A by the corresponding element in the matrix TF whose elements are the twiddle factors. Each of the 6 steps can be represented in general as a function S_i. We can now describe MO-FFT in powerlists as

$$\begin{aligned} MO-FFT << x >>&=<< x >> \\ MO-FFT(p) &= s6(s5(s4(s3(s2(s1(p)))))) \end{aligned} \tag{28}$$

where,

$$\begin{aligned} S_1(p) &= T(p) \\ S_2(A_1|A_2|...A_{n2}) &= F(A_1)|F(A_2)|...|F(A_{n2}) \\ S_3(p) &= P * TF \\ S_4(p) &= T(p) \\ S_5(A_1|A_2|...A_{n2}) &= F(A_1)|F(A_2)|...|F(A_{n1}) \\ S_6(p) &= T(p) \\ F(p) &= MO-FFT(p) \end{aligned} \tag{29}$$

From S_2 and S_5, it can be seen that it involves computing MO-FFT recursively on each of the n_2 rows of matrix A. Since these are independent of each other, all the n_2 computations of MO-FFT can be computed in parallel using parallel-for. Again for S_4, it is just a scalar operation applied to each of the elements of the matrix i.e each of the n elements are multiplied by the corresponding twiddle factor, which again, can be computed in parallel. Although Matrix Transposition is recursive, it too offers parallelism as is evident from Equation 26.

Thus, we have seen various cache-oblivious and multicore-oblivious algorithms and how they can be easily specified using powerlist notations.

5.6 Discussion

We have shown how powerlist notation can be an efficient way to transform algorithms on to the multi-core architectures including GPUs through an illustration of the matrix multiplication problem. The important aspect of demonstration has been the use of powerlists to specify both parallelism and recursion at the same time; this leads to methods to identify optimal depth of recursion based on the multi-core parameters which helps in determining the execution configuration parameters best suited for the architecture. We have also shown how the abstraction provided by powerlist allows us to automate the partitioning and of problems (and merging) of subproblems by eliminating the need for the user to partition the matrices. This approach has also been extended to realize FFT, and can be adapted to any application that is recursive in nature. We have also highlighted the relationship between powerlists and oblivious algorithms, having demonstrated how various oblivious algorithms can be succinctly expressed using powerlists. Thus, powerlists can be used as a high level abstraction to specify parallelism within an application, and giving its implementation the responsibility of exploiting the parallelism in the best possible way, thereby offloading much of the burden from the programmer.

Chapter 6

Conclusions and Future Scope

In this work, we have arrived at paradigms to provide solutions to problems pertaining to the challenges of multi-core programming. The first part of our work based on STMs strives to address performance and productivity concerns for concurrent systems. From the experiments we see that the performance of CaPR+ for the kmeans, genome, and ssca2 applications suggests that the partial rollback mechanism fares better than the abort mechanism in cases, where the transaction lengths are relatively longer. This led us to experiment by modifying these applications to have transactions of varying transactional lengths. The comparative evaluation of Abort and partial rollback in the presence of local transactional delays concludes that partial rollback is better-off only for applications with large transactions and high contention among threads, which suggests a partial rollback based STM may not be practical for most applications. This result helped us to arrive at the integrated partial rollback-abort framework, that exploits the benefits of both mechanisms.

Our work on powerlists addresses performance and productivity for parallel systems. It demonstrates how powerlists can be exploited as a high-level language by using it to just specify recursion and parallelism, and letting the runtime exploit the parallelism in the best possible way. In particular, we have demonstrated the use of powerlists to specify computations succinctly, and how the parameters of multi-core architectures, in particular GPUs, can be effectively utilized to predict the optimal kernel execution parameters. We have also described a method to schedule matrix multiplication across a cluster of GPUs, and experimentally shown the performance gain achieved is nearly 132% over the computation on a single GPU using CUBLAS library. The performance gain is more for larger matrices. We have also shown how the computation could be scaled for larger matrices which could not have been computed using CUBLAS on a single GPU. Although we have demonstrated our approach for matrix multiplication, it can also be applied to a broad range of computations beyond matrix multiplication. This would involve the whole class of applications that are recursive in nature, and can be expressed in the powerlist notation like FFT, Prefix-sum, Batcher-sort etc.

We have also provided implementations of two deadlock-free lock-based synchronization mechanisms to address the reliability concerns. Par-Deadlockfree has the potential to considerably expand the scope of GPGPU computing by providing efficient support for fine-grained inter-block synchronization that is also deadlock-free. Further, we have shown how Par-Deadlockfree can be used for implementing TMs, that would considerably improve the programmers' productivity, while also handling the synchronization appropriately.

Future Works

It would be nice to explore the possibility of adapting machine learning for scheduling purposes similar to that envisaged in the work of Narang [124] to realize a dynamic implementation of the hybrid approach discussed in Section 3.3, wherein we can dynamically switch between mechanisms, based on the transaction length and other parameters. We are also working towards realizing a Software Transactional Memory (STM) for GPUs that internally uses Par-Deadlockfree for synchronization. This framework would simultaneously address all the three challenges: Performance, Productivity and Software Reliability, and is expected to considerably widen the scope of GPGPU computing. Further, we have illustrated in Section 5.5 how powerlists can be used to succinctly specify cache-oblivious and multicore-oblivious algorithms. This abstraction provided by powerlists can be leveraged to realize a high-level framework to specify recursive algorithms, leaving the responsibility of exploiting the parallelism in the best possible way to its implementation, thereby offloading much of the burden from the programmer.

Bibliography

[1] S. Amarasinghe. The Looming Software Crisis due to the Multicore Menace. *lecture Nati'l Science Foundation*, 2007, [online] Available: http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf.

[2] Amdahl, G M. Validity of the single processor approach to achieving large scale computing capabilities. *In Proc. Spring Joint Computer Conference*, pp.483485, ACM (1967).

[3] R. Dennard, et al.. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid State Circuits*, vol. SC-9, no. 5, pp. 256-268, Oct. 1974.

[4] http://www.itrs2.net/itrs-reports.html. *The International Technology Roadmap for Semiconductors 2.0*, 2015.

[5] R. K. Shyamasundar. Multicore and Extreme Scale Computing: Programming and Software Challenges. Keynote lecture, *International Conference on Advances in ICT for Emerging Regions (ICTER)*, Colombo, 2012.

[6] Anshu S Anand, R. K. Shyamasundar, and Sathya Peri. Opacity Proof for CaPR+ Algorithm. *International Conference on Distributed Computing and Networking, ICDCN 2016*, Singapore, DOI: 10.1145/2833312.2833445.

[7] Anand AS, Shyamasundar RK, and Peri S. STMs in practice: Partial rollback vs pure abort mechanisms. *Concurrency Computat Pract Exper. 2018*;e4465. https://doi.org/10.1002/cpe.4465.

[8] Rachid Guerraoui and Michael Kapalka. Principles of Transactional Memory. *Morgan & Claypool publishers*, 2010.

[9] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory. 2nd edition, *Morgan & Claypool publishers*, 2010.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In Proc. of the *Twentieth Annual International Symposium on Computer Architecture*, pp. 289-300, San Diego, California, 1993, ACM Press.

[11] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing*, Special Issue (10):99-116, 1997.

[12] S. Kumar, M. Chu, et al. Hybrid transactional memory. In Proceedings of the eleventh *ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP 06*, ACM Press, New York, NY, USA, March 2006.

[13] http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/

[14] Joo Cachopo and Antnio Rito-Silva. Versioned boxes as the basis for memory transactions. SCOOL05: Proc. *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[15] Aleksandar Dragojevic, Rachid Guerraoui, and Michael Kapalka. Stretching transactional memory. In PLDI 09: Proc. 2009 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 155-165, June 2009.

[16] http://www.alphaworks.ibm.com/tech/xlcstm.

[17] Koskinen E and Herlihy M. Checkpoints and continuations instead of nested transactions. In Proceedings of the *Twentieth annual symposium on Parallelism in algorithms and architectures (SPAA08)* (New York, NY, USA, 2008), ACM, pp. 160-168, 2008.

[18] McKenney, P.E. Is parallel programming hard, and, if so, what can you do about it? (2013); https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[19] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In Proceedings of the *20th International Symposium on Distributed Computing (DISC06)*, 2006.

[20] P. A. Bernstein and N. Goodman. Multiversion concurrency control theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465-483, 1983.

[21] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software trans- actional memory. In *TRANSACT 06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[22] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In Proceedings of the *13th ACM SIGPLAN Symposium on Principles and practice of parallel program- ming, PPoPP 08*, pp. 175-184. ACM, 2008.

[23] D. Dice, O. Shalev, et al. Transactional locking II. In *DISC06: Pro- ceedings of the 20th International Symposium on Distributed Computing*, March 2006.

[24] V. J. Marathe, M. F. Spear, et al. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[25] A. Jaleel, M. Mattina, et al. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. In the *Twelfth International Symposium on High-Performance Computer Architecture*, 2006, pp. 88-98, 11-15 Feb. 2006. ISSN 1530- 0897.

[26] R. Narayanan, B. Ozisikyilmaz, et al. Minebench: A benchmark suite for data mining workloads. In *IEEE International Symposium on Workload Characterization*, pp. 182-188, Oct. 2006.

[27] Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A, Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., and Yazawa, K. The design and implementation of a first-generation CELL processor. In *Solid-State Circuits Conference, 2005*, Digest of Technical Papers. ISSCC pp.184,592 Vol. 1, 10-10 Feb. 2005.

[28] Hofstee, H.P. Power efficient processor architecture and the cell processor. In *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11*, pp.258,262, 12-16 Feb. 2005

[29] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *SIGPLAN Not. 40*, 519-538, October 2005.

[30] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In Proceedings of the *12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)*, ACM, New York, NY, USA, 183-193, 2007.

[31] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proceedings of the *2008 ACM/IEEE conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 31 , 11 pages, 2008.

[32] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)* ACM, New York, NY, USA, 73-82, 2008.

[33] Jacob Kornerup. Mapping Powerlists onto Hypercubes. Technical Report, *University of Texas at Austin*, Austin, TX, USA, 1994

[34] David S. Wise. Representing matrices as quadtrees for parallel processors: Extended abstract. *SIGSAM Bull.*, 18(3):24-25, August 1984.

[35] NVIDIA Corporation: NVIDIA CUDA C best practices guide. 2013. URL *http : //docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices$_G$uide.pdf*. Version 5.5.

[36] NVIDIA Corporation: NVIDIA CUDA C programming guide. 2013. URL http:// docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Version 5.5.

[37] NVIDIA Corporation: NVIDIA fermi compute architecture whitepaper. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Comp ute_Architecture_Whitepaper.pdf.

[38] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. In *PARALLEL COMPUTING*, 27:2001, 2000.

[39] M. Frigo and S.G. Johnson. The design and implementation of fftw3. In Proceedings of the *IEEE*, 93(2):216-231, Feb 2005.

[40] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms. In *ACM, PLDI 01*, pp. 298-308, 2001.

[41] Petr Kuznetsov and Sathya Peri. On non-interference of transactions. CoRR, abs/1211.6315, 2012

[42] Koskinen E and Herlihy M. Checkpoints and continuations instead of nested transactions. In Proceedings of the *Twentieth annual symposium on Parallelism in algorithms and architectures (SPAA08)* (New York, NY, USA, 2008), ACM, pp. 160-168, 2008.

[43] Lupei, D. A study of conflict detection in software transactional memory. Masters thesis, *University of Toronto*, the Netherlands, 2009.

[44] Gupta, M., Shyamasundar, R.K., and Agarwal, S. Article: Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. In *International Journal of Computer Applications* 1(22), 80-85, February 2010.

[45] Gupta, M., Shyamasundar, R.K., and Agarwal, S. Automatic checkpointing and partial rollback in software transaction memory (January 2012) *US Patent* 20110029490.

[46] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. In ACM *Transactions on Programming Languages and Systems*, 12(3):463-492, June 1990.

[47] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In Proceedings of the *20th International Symposium on Distributed Computing (DISC06)*, 2006.

[48] P. A. Bernstein and N. Goodman. Multiversion concurrency control theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465-483, 1983.

[49] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, TRANSACT 06*, June 2006.

[50] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In Proceedings of the *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP 08*, pp. 175-184, ACM, 2008.

[51] D. Dice, O. Shalev, et al. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC06*, March 2006.

[52] V. J. Marathe, M. F. Spear, et al. Lowering the overhead of nonblocking software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, TRANSACT'06*, 2006.

[53] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315-324, 2007.

[54] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In Proceedings of the *22nd International Symposium on Computer Architecture*, pp. 24-36. June 1995.

[55] Standard Performance Evaluation Corporation, SPEC OpenMP Benchmark Suite. http://www.spec.org/omp.

[56] A. Jaleel, M. Mattina, et al. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. *The Twelfth International Symposium on High-Performance Computer Architecture*, 2006, pp. 88-98, 2006.

[57] R. Narayanan, B. Ozisikyilmaz, et al. Minebench: A benchmark suite for data mining workloads. *2006 IEEE International Symposium on Workload Characterization*, pp. 182-188, Oct. 2006.

[58] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC 08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[59] M. M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS'08*, pp. 1-11, IEEE Computer Society, 2008.

[60] Porfirio Alice et. al. Transparent Support for Partial Rollback in Software Transactional Memories. In *Euro-Par'13*, 2013.

[61] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *OPODIS'11*, pp. 112-127, 2011.

[62] Hagit Attiya and Eshcar Hillel. A single-version stm that is multi-versioned permissive. *Theory Comput. Syst.*, 51(4):425-446, 2012.

[63] Rachid Guerraoui and Michal Kapalka. Principles of Transactional Memory. *Synthesis Lectures on Distributed Computing Theory*, Morgan and Claypool, 2010.

[64] Tyler Crain, Damien Imbs, and Michel Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In *ICA3PP (1)*, pp. 244-257, 2011.

[65] Damien Imbs and Michel Raynal. A lock-based stm protocol that satisfies opacity and progressiveness. In Proceedings of the *12th International Conference on Principles of Distributed Systems, OPODIS 08*, pp. 226-245, Springer-Verlag, Berlin, Heidelberg, 2008.

[66] Dice D., Shalev O. et al: Transactional locking II., In Proceedings of the *20th International Symposium on Distributed Computing, DISC06*, 2006.

[67] Cachopo, J. and Rito-Silva, A. Versioned Boxes as the Basis for Transactional Memory. *Science of Computer Programming*, 63(2): 172-175, 2006.

[68] Riegel, T., Felber, P. and Fetzer, C. A Lazy Snapshot Algorithm with Eager Validation. In *DISC'06*, LNCS, vol. 4167, Springer, Heidelberg, pp. 284-298, 2006.

[69] Scherer W. N. III and Scott M. L. Advanced contention management for dynamic software transactional memory. In Procs of the *24th Annual ACM Symposium on Principles of Distributed Computing (PODC '05)*, ACM, New York, USA, pp. 240-248, 2005.

[70] Guerraoui R., Herlihy M., and Pochon B. Toward a theory of transactional contention managers. In Procs of the *24th Annual ACM Symposium on Principles of Distributed Computing (PODC '05)*, ACM, New York, USA, pp. 258-264, 2005.

[71] Spear M. F. , Dalessandro L., Marathe V. J., and Scott M. L. A comprehensive strategy for contention management in software transactional memory. In Procs

of the *14th ACM SIGPLAN Symposium on Principles and Practice Of Parallel Programming (PPoPP '09)*, ACM, New York, USA, pp. 141-150, 2009.

[72] Gerhard Weikum and Gottfried Vossen. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. *Morgan Kaufmann*, 2002.

[73] Threading Building Blocks, http://threadingbuildingblocks.org.

[74] James Reinders. Intel Threading Building Blocks. (First ed.). *OŔeilly & Associates*, Inc., Sebastopol, CA, USA ,2007.

[75] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *workshop. on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Ottawa, ON, Canada, June 2006.

[76] Khronos OpenCL Working Group. The OpenCL Specification Version 1.1. *Khronos Group*, 2009. http://www.khronos.org/opencl. Online.

[77] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science and engineering*, 2010.

[78] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-world Applications. In Procs of the *18th International Conference on Parallel Processing, Euro-Par'12*, pp. 859-870, Springer-Verlag, Berlin, Heidelberg, 2012.

[79] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. IEEE Trans. Parallel Distrib. Syst., vol. 1, no. 1, pp. 6-16, Jan. 1990.

[80] D. Cederman and P. Tsigas. Dynamic Load Balancing using Work-Stealing. In *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Elsevier Science, pp. 485-499, 2011.

[81] A. Yilmazer and D. Kaeli. HQL: A Scalable Synchronization Mechanism for GPUs. In Proc. of *27th IEEE International Symposium on Parallel Distributed Processing (IPDPS'13)*, 2013.

[82] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting finegrained synchronization on a simultaneous multithreading processor. In Proceedings of *Fifth International Symposium on High-Performance Computer Architecture*, pp. 54-58, jan 1999.

[83] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware transactional memory for GPU architectures. In Proceedings of the *44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 11*, New York, NY, USA: ACM, pp. 296-307, 2011.

[84] Xu, Yunlong, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. Lock-based synchronization for GPU architectures. In Proceedings of the *ACM International Conference on Computing Frontiers*, pp. 205-213, ACM, 2016.

[85] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software Transactional Memory for GPU Architectures. In Proc. of *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[86] Anand AS, Srivastava A, Shyamasundar RK. A deadlock-free lock-based synchronization for GPUs. *Concurrency Computat Pract Exper.*, 2018;e4991. https://doi.org/10.1002/cpe.4991.

[87] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In Proc. of the *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[88] Wilson Wai Lun Fung. GPU Computing Architecture for Irregular Parallelism. PhD Thesis, *The University of British Columbia*, January 2015.

[89] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In Proceedings of the *IEEE International Parallel and Distributed Processing Symposium, IPDPS'07*, pp. 110, IEEE, 2007.

[90] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In Proc. of the *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 73-82, Feb. 2008.

[91] A. Ramamurthy. Towards Scalar Synchronization in SIMT Architectures. Master's thesis, *University of British Columbia*, 2011.

[92] A. Li, G. Braak, H. Corporaal, and A. Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In Proc. of the *29th ACM on International Conference on Supercomputing, ICS'15*, 2015.

[93] Sarnath. Spinlock on a GPU. http://forums.nvidia.com/index.php?showtopic=98444&st=40/, June 2009.

[94] NVIDIA. PTX: Parallel Thread Execution ISA Version 4.3. http://docs.nvidia.com/cuda/ parallel-thread-execution/index.html.

[95] Brett W Coon, Peter C Mills, John R Nickolls, and Lars Nyland. Lock mechanism to enable atomic updates to shared memory, US Patent 8,055,856, Nov 8, 2011.

[96] Coffman, E.G., M.J. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67-78, 1971.

[97] Lee C. Y. An algorithm for path connections and its applications. In *IRE Transactions on Electronic Computers*, EC-10(3): pp. 346-365, 1961.

[98] Anshu S Anand and R. K. Shyamasundar. Scaling Computation on GPUs Using Powerlists. In *Foundations of Big Data Computing*, in conjunction with HiPC-2015, Bangalore, DOI: 10.1109/HiPCW.2015.14.

[99] NVIDIA Corporation. CUBLAS LIBRARY User Guide, February 2014. Version 6.0.

[100] Cooley, James W. and Tukey, John W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19: pp. 297301, 1965.

[101] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (2nd ed.). *McGraw-Hill Higher Education*, 2001.

[102] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107-113, January 2008.

[103] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Trans. Program. Lang. Syst.*, 16(6):1737-1767, November 1994.

[104] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In Procs of the *ACM/IEEE Conference on Supercomputing, SC '08*, Austin, TX, pp. 1-12, 2008.

[105] Jacob Kornerup. Data Structures for Parallel Recursion. PhD thesis, *University of Texas at Austin*, 1997.

[106] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pp. 270-289, 2013.

[107] Coffman, E.G., Elphick, M. and Shoshani, A. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2), pp.67-78, 1971.

[108] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Mndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pp. 12-25, 2011.

[109] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on GPUs. In Proceedings of the *18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13)*, ACM, New York, NY, USA, 147-156, 2013.

[110] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In Proceedings of the *IEEE International Symposium on Workload Characterization, IISWC '12*, IEEE Computer Society, Washington, DC, USA, 141-151, 2012.

[111] J. Reinders. Transactional synchronization in haswell. *Intel Developer Zone*, February 2012.

[112] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In Proceedings of the *40th Annual International Symposium on Computer Architecture*, ISCA 13, pp. 225-236, ACM, New York, NY, USA, 2013.

[113] H.Q. Le, G.L. Guthrie, D.E. Williams, M.M. Michael, B.G. Frey, W.J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1), 2015.

[114] C.C Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *MICRO-45*, 2012.

[115] Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place. In *WTTM12*, 2012.

[116] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[117] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, pp. 178-198, June 1978.

[118] Edsger Wybe Dijkstra. Cooperating Sequential Processes, *Technical Report Ewd-123*, Technical Report, 1965.

[119] Damien Imbs and Michel Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoret-*

ical Computer Science, Volume 444, pp. 113-127, June 1978. ISSN: 0304-3975, https://doi.org/10.1016/j.tcs.2012.04.037.

[120] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In Proc. of *FOCS'99*, pp. 285-297, 1999.

[121] Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri and Brandon Blakeley. Oblivious algorithms for multicores and networks of processors, In *Journal of Parallel and Distributed Computing (JPDC)*, Volume 73, Issue 7, pp. 911-925, 2013. ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2013.04.008.

[122] Holt R. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

[123] V. Volkov and J.W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In Proc. of *Intl Conf. High Performance Computing, Networking, Storage and Analysis*, pp. 1-11, 2008.

[124] Narang A, Srivastava A, Shyamasundar RK. High performance adaptive distributed scheduling algorithm. In *IEEE International Symposium on Parallel and Distributed Processing*, Workshops and PhD Forum, Cambridge, MA, 2013.