Hybrid Multicore Algorithms for Some Semi-Numerical Applications and Graphs

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

DOCTOR OF PHILOSOPHY
in
Computer Science and Engineering

by

Dip Sankar Banerjee
200907007
dipsankar.banerjee@research.iiit.ac.in

Center for Security, Theory, and Algorithmic Research
International Institute of Information Technology
Hyderabad - 500 032, INDIA
December, 2014

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Hybrid Multicore Algorithms for Some Semi-Numerical Applications and Graphs" by Dip Sankar Banerjee, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Dr. Kishore Kothapalli
Date	Dip Sankar Banerjee

To My Grandfather

Late Shri Mono Mohan Mukherjee

Acknowledgments

I want to express immense gratitude to my advisor Dr. Kishore Kothapalli for his excellent guidance, support and motivation. His deep insights into all of the problems tackled in this thesis combined with exemplary work ethics has truly made this journey an enjoyable one.

I want to thank the members of my thesis committee comprising of Prof. R Govindarajan from the Indian Institute of Science, Dr. Kannan Srinathan and Dr. Suresh Purini for their valuable feedbacks regarding several aspects of this thesis.

I want to thank my teachers, Prof. PJ Narayanan, Prof. R. Govindarajulu who has endowed on me knowledge that I will treasure upon for the rest of my life. I also wish to thank my research collaborators Pariskshit Sakurikar, Shashank Sharma and Aman Bahl. They provided immense support towards the completion of some of the critical components of the thesis and it was an absolute pleasure working with them.

It was a excellent opportunity to work with all my lab mates at the Center for Security, Theory and Algorithmic Research (C-STAR) over the last four years and I want to thank each one of them for making my stay and work here so easy. Several of my friends at the International Institute of Information Technology, Hyderabad provided me an excellent atmosphere to dwell, think, work and also to spend fun times. Thanks to Broto Chakrabarty, Shamba Shankar Mandal, Akshat Kumar and Swarnabha Sen for all the memories.

I want to take this opportunity to thank my parents whose endless examples of pursuing excellence motivated me to work hard and stay focused at all times. I also express my happiness to my loving wife Suchetana Chakraborty who has made everything possible in so many ways.

Abstract

The computing industry has undergone several paradigm shifts in the last few decades. Fueled by the need of faster computing, larger data and real time processing needs parallel computing has emerged as one of the dominant paradigms. Motivated by the success achieved in distributed computing models and the limitations faced by single core processors, parallel computing is the only alternative for building faster computers. Parallel computing is one of the most challenging areas computer science in the present and developing algorithms and optimization techniques for utilizing the processing power present in a current generation parallel computer is still a very exciting area for research.

The parallel computing industry underwent a massive shift with the conventional sequential computers hitting the power wall. It led to the development of multicore and many-core computing chips that had multiple sequential computing cores packed into a single chip. The immediate impact was the need for (re)designing sequential algorithms in order to utilize the computing power of such chips. Combined with the intricate memory and cache structures, parallel algorithms require a higher degree of engineering for the most optimal performance.

The many-core revolution started with the release of Graphics Processing Units (GPU) which had a large number of compute cores and offered massive parallelism. With the evolution of the many-core chips, the GPUs found application in graphics, gaming as well as general purpose computation. In the same time frame, the Central Processing Units (CPU) too under went a sea of innovation and emerged as more powerful and mature computing machines. However, the multicore CPUs were mostly ignored in its initial days. With the advancement of accelerator platforms, the CPUs and GPUs are now able to communicate in a more efficient manner. In the recent times there has been quite a few works such as the ones in [79, 91, 43] that shows that hybrid algorithms actually provide better performance and efficiency over conventional accelerator based computing.

In this thesis we work towards the development of *hybrid multicore computing*. Hybrid multicore computing is developing algorithms and optimization strategies for popular computing primitives on an hybrid platform. A hybrid platform is one which contains two or more multicore or many-core devices. There are several challenges towards the efficient algorithm design on hybrid platforms such as that of communication bottlenecks, load balance and synchronization. In this thesis we work towards developing algorithms for some computing primitives such as that of list ranking, sorting and pseudo-randomness and some graph algorithms. We experiment on a hybrid platform consisting of a 6-core Intel CPU and Nvidia GPUs of broadly two generations.

In our first work we work towards the development of hybrid algorithms for List Ranking. To this end, we explore the algorithmic and analytical issues in hybrid multicore computing. Our case studies involve two different ways of designing hybrid multicore algorithms. The main contribution of this work is to address the issues related to the design of hybrid solutions. We show our hybrid algorithm for list ranking is faster by 50% compared to the best known implementation [139].

This work is followed by a hybrid implementation of comparison sorting. Sorting has been a topic of immense research value since the inception of Computer Science. In this work, we present a hybrid comparison based sorting algorithm which utilizes a many-core GPU and a multi- core CPU to perform sorting. The algorithm is broadly based on splitting the input list according to a large number of splitters followed by creating independent sublists. Sorting the independent sublists results in sorting the entire original list. On a hybrid platform, our algorithm achieves a 20% gain over the current best known comparison sort result that was published by Davidson et. al. [32]. On the above experimental platform, our results are better by 40% on average over a similar GPU-alone algorithm proposed by Leischner et. al. [81]. Our results also show that our algorithm and its implementation scale with the size of the input. We also show that such performance gains can be obtained on other hybrid platforms.

The use of many-core architectures and accelerators, such as GPUs, with good programmability has allowed them to be deployed for vital computational work. The ability to use randomness in computation is known to help in several situations. For such computations to be made possible on a general purpose computer, a source of randomness, or in general a pseudo random generator (PRNG), is essential. However, most of the PRNGs currently available on GPUs suffer from some basic drawbacks. It is of high interest to develop a parallel, quality PRNG that also works in an on demand model. In this work, we investigate a hybrid technique to create an efficient PRNG. The basic technique we apply is that of random walks on expander graphs. Unlike existing generators available in the GPU programming environment, our generator can produce random numbers on demand as opposed to a one- time generation. Our approach produces 0.07 GNumbers per second. The quality of our generator is tested with industry standard tests.

In the second part of the thesis, we work towards hybrid graph connected components and breadth first search. For computing graph connected components we use a static load balancing technique for partitioning of work between two devices and is followed by a low overhead consolidation phase for merging the results. We achieve almost 25% improvement over the best known implementation in [123]. For performing breadth first search (BFS), we employ a graph pruning strategy to reduce the size of large graphs which is often composed of a large percentage of pendant nodes. We apply a hybrid BFS on the remainder graph and is followed by a re-insertion phase. We achieve a 35% improvement over the current best know solution which was published by Munguia et al. in [91].

Contents

Chapter Page

1 **Introduction** . . . 1
- 1.1 Parallel Computing Background . . . 1
  - 1.1.1 Need for Parallel Computing . . . 1
  - 1.1.2 Models of Parallel Computing . . . 1
  - 1.1.3 Large Scale Supercomputing . . . 2
- 1.2 Accelerator based Computing . . . 3
  - 1.2.1 Graphics Processing Units (GPU) . . . 5
  - 1.2.2 IBM Cell Broadband Engine . . . 6
  - 1.2.3 Some other works using accelerators . . . 7
- 1.3 Beyond Accelerator based computing . . . 8
  - 1.3.1 Non-overlapped hybrid computing . . . 9
  - 1.3.2 Overlapped hybrid computing . . . 10
  - 1.3.3 Current Results in Hybrid Computing . . . 11
- 1.4 Summary of this thesis . . . 12

2 **GPU and CPU Parallelism** . . . 13
- 2.1 Introduction . . . 13
- 2.2 Multicore and Manycore . . . 14
- 2.3 GPU Architecture . . . 15
- 2.4 GPGPU . . . 18
- 2.5 Compute Unified Device Architecture (CUDA) . . . 19
- 2.6 Multicore CPU . . . 22
- 2.7 Open Multi-Processing (OpenMP) . . . 25
- 2.8 Open Computing Language (OpenCL) . . . 27
- 2.9 Hybrid Platform . . . 28
  - 2.9.1 Hybrid High . . . 28
  - 2.9.2 Hybrid Low . . . 29

3 **Our Contributions** . . . 30
- 3.1 Motivation . . . 30
- 3.2 Key Targets . . . 31
- 3.3 Parallel Comparison Sorting . . . 32
- 3.4 Hybrid Pseudo Random Number Generator . . . 34
- 3.5 Parallel List Ranking . . . 35
- 3.6 Graph Connected Components . . . 36

3.7 Breadth First Search 38

I Semi-Numerical Algorithms **41**

4 **Hybrid Comparison Sorting** 44
4.1 Introduction 44
4.1.1 Motivation 45
4.1.2 Related Work 46
4.2 Our Solution 47
4.2.1 Phase I 49
4.2.2 Phase II 49
4.2.3 Phase III 50
4.2.4 Phase IV 50
4.3 Implementation Details 51
4.3.1 Phase I 51
4.3.2 Phase II 52
4.3.3 Phase III 53
4.3.4 Phase IV 53
4.3.5 Memory Usage 54
4.3.5.1 Higher coalescing of reads 55
4.3.5.2 Reuse of histogram store 55
4.3.6 Sorting variable length keys 55
4.4 Experiments and Results 56
4.4.1 Profiling, Resource Utilization, and Idle Times 58
4.4.2 Results of Sorting 59
4.4.2.1 Results on Fixed Length Keys 59
4.4.2.1.1 64 Bit Inputs 61
4.4.2.1.2 Results on Other Platforms 62
4.4.2.2 Variable Length Key Sorting 63
4.4.3 Threshold Variation 65
4.4.4 Scalability 65
4.4.5 Results of Phase II 65
4.5 Conclusion 66

5 **Pseudo Random Number Generator for Hybrid Platforms** 67
5.1 Introduction 67
5.1.1 Motivation 67
5.2 Related Work 69
5.2.1 Our Methodology and Results 69
5.3 Our Random Number Generation Technique 69
5.3.1 Expander Graphs 70
5.3.2 Implementation Details 71
5.4 Experimental Results 73
5.4.1 Performance Analysis 73
5.4.2 Quality 76
5.4.3 Discussion 77

5.5 Application : Hybrid Monte Carlo . . . 77
5.5.1 Our solution . . . 78
5.6 Conclusions . . . 80

II Graph Algorithms **81**

6 **Parallel List Ranking** . . . 84
6.1 Introduction . . . 84
6.1.1 Related Work . . . 84
6.1.2 Our Results . . . 85
6.2 Recursive Hellman Jaja Algorithm for List Ranking . . . 86
6.2.1 CUDA implementation of RHJ . . . 87
6.3 Hybrid List Ranking . . . 87
6.3.1 The Proposed Solution . . . 88
6.3.2 Fractional Independent Sets . . . 88
6.4 Computing FIS using PRNG from Chapter 5 . . . 91
6.4.1 Experimental Results . . . 92
6.4.1.1 Results using PRNG from Chapter 5 . . . 96
6.5 Conclusions . . . 97

7 **Hybrid Graph Connected Components** . . . 98
7.1 Introduction . . . 98
7.2 Related Work . . . 98
7.3 The Shiloach Vishkin Algorithm . . . 99
7.4 A Hybrid Algorithm for Graph Connected Components . . . 99
7.4.1 Example . . . 101
7.5 The Shiloach Vishkin Algorithm for GPU . . . 101
7.6 DFS on CPU . . . 102
7.7 Processing Cross Edges . . . 102
7.8 Results . . . 103
7.9 Static Auto-tuning . . . 104
7.10 Conclusions and Future Work . . . 105

8 **Parallel Breadth First Search** . . . 107
8.1 Introduction . . . 107
8.1.1 Related Work . . . 108
8.1.2 Our Results . . . 109
8.2 Our Approach . . . 110
8.3 Breadth First Search . . . 112
8.3.1 Implementation . . . 112
8.3.2 Phase I . . . 113
8.3.3 Phase II . . . 113
8.3.4 Phase III . . . 116
8.3.5 Results . . . 116
8.4 Connected Components . . . 119
8.4.1 Implementation . . . 119

8.4.2 Results 121
8.5 Conclusions 122

9 **Conclusions and Future Directions** 123
9.1 Conclusions 123
9.2 Future Directions 124
9.2.1 All Pairs Shortest Path 124
9.2.2 Randomized Approaches 125

Bibliography 127

List of Figures

Figure		Page
1.1	The PRAM Model.	2
1.2	Floating point operations per second for the CPU and GPU. Plot obtained from freely available documents provided by Nvidia.	4
1.3	Conventional model of accelerator based computing.	5
1.4	The Cell BE Processor. Image obtained from Cell BE Programming Guide distributed by IBM.	6
1.5	Non-overlapped hybrid execution.	10
1.6	Overlapped hybrid execution.	11
2.1	Performance gap between CPUs and GPUs.Image obtained from [71].	15
2.2	Basic GPU Architecture.	16
2.3	A Nvidia Tesla 4 GPU, Nvidia GTX 580 GPU and a GTX 600 Series GPU. The images are obtained from freely available documents provided by Nvidia.	16
2.4	Shader Model 4.0 Pipeline. Geometry shader was introduced as new programmable unit in the pipeline.	19
2.5	CUDA Hardware Model.	20
2.6	CUDA Software Model.	21
2.7	OpenCL device architecture. Image obtained from OpenCL Specification.	27
3.1	Performance on key-value pairs in high-end platform.	33
3.2	Performance on key-value pairs in low-end platform.	33
3.3	Performance over variation of sizes generated.	35
3.4	Improvement of List Ranking from 2009 to the current hybrid implementation. We can see that we acheived a speedup of mearly 3x over the original GPU implementation.	36
3.5	Time comparison of connected components with respect to the results shown by Soman et al. in [123].	37
3.6	Performance of BFS on the UFL graphs obtained from [2]. The percentages indicate the improvement over the results of [91] over the same instance.	38
4.1	Different phases in hybrid sorting. The different colors represent the different bin labels which are brought together by scattering.	48
4.2	An example run of our algorithm on a sample input of 9 elements. In this, we show the first pass of the algorithm on our input list, that creates the first set of bins.	48

4.3 The count for each bin label is computed through histograms in shared memory and is written back in column major order to global memory. Then the prefix sum provides the offsets for each of the labels. 54
4.4 Phase wise timing diagram for 4 million elements. 57
4.5 Percentage improvement over sample sort [81] at various phases. 58
4.6 Performance on uniformly random keys sorting in high-end platform. 59
4.7 Performance on key-value pairs in high-end platform. 60
4.8 Performance on 32 bit Gaussian input. 60
4.9 Performance on 32 bit Deterministic Duplicates input. 61
4.10 Performance on 32 bit staggered input. 61
4.11 Performance on 32 bit bucket sorted input. 61
4.12 Performance on 32 bit randomized duplicates input. 61
4.13 Performance on 64 bit gaussian input. 62
4.14 Performance on 64 bit deterministic duplicates input. 62
4.15 Performance on key value pairs on low-end platform. 62
4.16 Performance of 32 bit sorting across different platforms at input of 2^{21} uniformly random elements. 63
4.17 Performance of key value sorting across different platforms at input of 2^{21} elements. . 63
4.18 Performance on variable length random strings. 64
4.19 Performance on protein database strings with caching. 64
4.20 Performance of variable length key sorting across different platforms. 64
4.21 Variation of threshold. 64
4.22 Time comparison with respect to pure GPU implementations and the performance gain. 66

5.1 The expander graph G_{13}. 71
5.2 Timings across several list sizes 74
5.3 The overlapped execution of the work units. 74
5.4 Variation of timing with block size. 75
5.5 The time comparison when the algorithm runs on the CPU vs CPU `rand()` 75
5.6 Variation of timing with number of photons simulated 79

6.1 The linked list and pre-processing done. (a) The initial list with the rank values in square brackets and random string in parenthesis. (b) Elements removed on the basis of the random string and ranks adjusted (c) List obtained after the pre-processing, (d) Ranking the remaining list, and (e) Restoration of the nodes removed from the list. . . 89
6.2 Time Comparison with respect to pure GPU implementations and the best known result. 93
6.3 Trade offs between Phase I and Phase II and the total timings for a 128M sized list . . 94
6.4 Work units in overlapped execution for a list of 128 M elements for the first 2 iterations. 95
6.5 The timing comparison with the other algorithms. 97

7.1 An example run of the algorithm. The red dashed lines denotes the partitions within the GPU or the CPU cores. The green edges are the cross-edges. 101
7.2 Time comparison with graphs sizes varying from 500K to 4.5M nodes. The GPU times are obtained by running the code from [123] on the same instances. 103
7.3 Time comparison in graphs of size 1.5 million and 3 million nodes. The GPU timings are the obtained by running the code from [123] on the same instances. 104
7.4 Threshold variation on graphs with sizes varying from 500K to 4.5M 106

7.5 Threshold variation on a graph of 2.5 million nodes and edge sizes varying from 3M to 8M 106

8.1 A sample of four real world graphs from [2]. On the top-left corner is the graph internet, top-right is the graph web-google, bottom left is the graph webbase_1M, and the bottom-right is the graph wiki-Talk. 108
8.2 The CSR format for representation. 112
8.3 An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III. 117
8.4 Performance of BFS on the UF Sparse Matrix dataset. The numbers show the percentage improvement. 117
8.5 Performance of BFS on the R-MAT dataset. The numbers show the percentage improvement. 117
8.6 Percentage improvement of performance of BFS as a function of the percentage of pendant nodes removed in Phase I. 118
8.7 Trade-off between Phase I and the overall runtime of BFS. 118
8.8 An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III. 120
8.9 Performance of Connected Components the UF Sparse Matrix dataset [2]. The numbers show the percentage improvement. 121
8.10 Performance improvement of Connected Components with the removal of pendant nodes. 122
8.11 Trade-off between Phase I and the total time. 122

List of Tables

Table Page

1.1 This table shows the accelerator results over known parallel implementations. Values in the brackets under Performance indicate the input size on which the performance was recorded. . . . 8
1.2 This table shows the hybrid results and the improvements over the accelerator results. The values in the brackets below the Performance indicates the input sizes on which the performance was recorded. The input element sizes are similar to that used in Table 1.1 for consistancy. . . . 12

3.1 Comparison of properties . . . 34

5.1 Comparison of properties . . . 69
5.2 Quality Results of different algorithms . . . 76
5.3 TestU01 test results . . . 77

6.1 Execution times across several list sizes. The list sizes are in million and the timings in milliseconds . . . 94
6.2 Generation, transfer and pre-processing timings for a list of 128 million. All the timings are in Milliseconds. Iteration 0 refers to the generation of the first batch of random numbers on the CPU and their transfer. . . . 95
6.3 Generation, transfer and pre-processing timings for a list of 16 million . . . 96

7.1 Timing for finding connected components and cross-edge processing. All timings are in ms and graph sizes in M . . . 105

8.1 The graphs used for experimentations and their properties. The column heading r in the last column indicates the number of iterations required to remove all pendant vertices. . 115

Chapter 1

Introduction

1.1 Parallel Computing Background

Computing paradigms have undergone a sea of change due to technological and economic considerations. Need for faster computation, requirements for handling larger volumes of data, real-time analysis and such have made the computing world undergo drastic paradigm shifts in several phases. Parallel computing is one of the major paradigm shifts that took place in the early parts of 1980s. The shift was motivated by the fact that in the near future, the Moore's Law of transistor packing in a single die will cease to hold. Also, there will be a power limitations to the clock frequency of a processor. Hence, it will be imperative to design distributed processors that will be required to work in unison to perform a certain computation.

1.1.1 Need for Parallel Computing

The biggest requirement for parallel computing was driven by the economy of world. There was a constant demand for faster and more capable machines. The economy had forced the creation of the "killer micro" [63]. It created a impediment towards innovative parallel computer design and was was thought of as a replacement of vector processors. Currently we see the massive scale improvement of many-core processors such as the Graphics Processing Units (GPU) and there is substantial proof to show the near future unification of the CPU and the GPU. Fundamental limitations of sequential processing such as the speed of light and dissipation of heat are the main causes [98].

1.1.2 Models of Parallel Computing

The earliest abstract machine of parallel computation was proposed by Fortune et al. in [40]. In this work, the authors proposed a shared memory parallel random-access machine (PRAM). A generic model of PRAM is shwon in Figure 1.1. In this model, the PRAM is intended as a parallel computing analogy to the random access machine (RAM) for the sequential algorithm designers. In a similar way where the RAM model neglects practical issues such as access time to cache memory versus the main memory, the

PRAM model also neglects some issues such as that of synchronization and communication. In order to identify the restrictions of the LogP model and make it more practical, there has been a wide array of works that have been produced such as the ones in [4, 5, 44, 67, 88, 101]. In another major work that re-works these neglects of the PRAM model was proposed by Culler et al. in the inspiring LogP model [30].

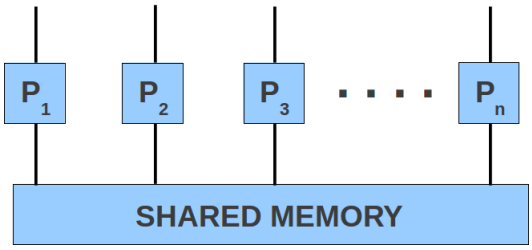

Figure 1.1 The PRAM Model.

Although the literature contains several taxonomies of parallelism [35, 53, 115], one can talk about two fundamental types of parallelism available for exploitation in software: data parallelism and task parallelism. Data parallelism is mostly concerned with the execution of the same instruction on a large data set. For example, element wise addition of two vectors such as the one that is commonly encountered in a lot of linear algebra applications commonly has a lot of data parallelism. Task parallelism on the other hand, is achieved by the decomposition of an application into several independent tasks. A multi-threaded web server can be a good example of task parallelism. Multiple requests of different nature are handled in parallel. In general, we can see that most of the applications in parallel computing use a combination of data and task parallelism. Therefore most of the applications can be placed somewhere in the middle of the entire spectrum. Although there might be a possibility to interchange between data and task parallelism for an already existing parallel application. This might not always be a possible.

In general, one can speak about a relationship between the current parallel architectures and types of available parallelism. For example, massively multithreaded architectures [38, 75] are better than others when dealing with large amounts of task parallelism. On the other side, GPUs [95] excel in data parallel computations. However, as most computations cannot be hard-classified as having solely task parallelism or solely data parallelism, an ultimate direct mapping of applications to architectures is an ambitious proposal.

1.1.3 Large Scale Supercomputing

In the decade from 1990-2000, the supercomputing market was mainly dominated by clusters made using off-the-shelf processors. Following that decade, of relative architectural stability we are currently in an era of several disruptions and divergences. There is a massive divergence in the nature of applications and hence there is a requirement for different resources. This further leads to diversity and

heterogeneity in architectures. Economics of scale on the other hand dictate only a handful of general purpose architectures that can be manufactured at the commodity level at low cost. On one hand there is the advantage of achieving low cost per unit on commodity processors, the custom made architectures can perfectly match with the underlying problems. Application Specific Integrated Circuits (ASIC) specifically falls under this custom made category of processors. Special purpose supercomputers, such as Anton [118], which is used to simulate molecular dynamics of biological systems, will still find applications where the reward exceeds the cost. For broader range of applicability, however, supercomputers that feature a balanced mixture of commodity and custom parts are likely to prevail.

Regardless of which architecture(s) will prevail in the end, the economic trends favor more parallelism in computing because building a parallel computer using large number of simple processors has proved to be more efficient, both financially and in terms of power, than using a small number of complex processors [117]. The software world has to deal with this revolutionary change in computing. It is safe to say that the software industry has been caught off-guard by this challenge. Most programmers are not fundamentally trained to think in parallel. Many tools, such as debuggers and profilers, that are taken for granted when writing sequential programs, were (and still are) lacking for parallel software development. In the last few years, there have been improvements towards making parallel programming easier, including parallel debugger, concurrency platforms, and various domain specific libraries.

One of the most promising approaches for tackling the software challenge in parallel computing is the top-down, application-driven, approach where common algorithmic kernels in various important application domains are identified. In the inspiring Berkeley Report [9], these kernels are called dwarfs(or motifs).

1.2 Accelerator based Computing

The multicore revolution was fueled by the fact that the standard processor designs collided with the power wall, the memory wall, and the ILP wall. Hence, the architectural designs progressed in the multi-core direction where several processing cores are packed into one single chip. Each of these cores run at a lower clock rate than a single core chip but the net throughput continues to rise. Another recent phenomenon is the availability of many-core accelerators such as the GPUs. It is because of the high cost to performance and the low cost to power ratios that has enabled GPUs become the devices of desktop supercomputing. In Figure 1.2, we can see the increase in the theoretical peak performance of the many-core GPUs and the multi-core CPUs over the last decade. Another similar architecture is the Cell Broadband Engine which was designed by IBM in 2001. The Cell BE was fundamentally an accelerator which had many heterogeneous features. In recent days architectures along the lines of accelerators continue to be released. The Intel Many Integrated Core(MIC) is one of the significant ones. However, the future roads in HPC systems are leading towards on-die hybrid systems at the level of supercomputing as well as the commodity consumers. It is evident from these developments that

the combination of general purpose CPU cores along with massively parallel accelerator cores provide higher performance benefits.

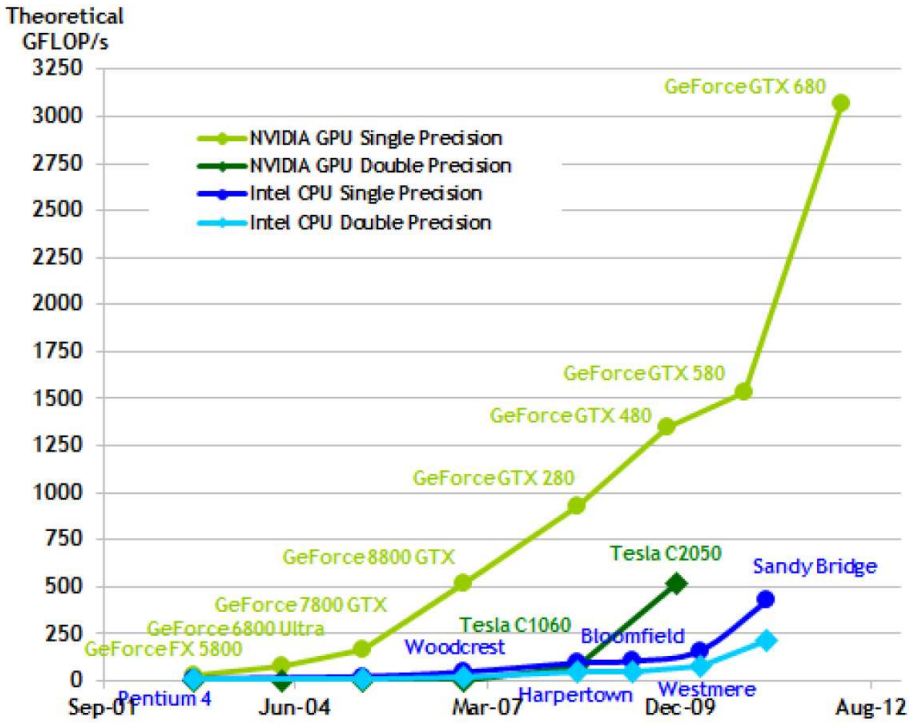

Figure 1.2 Floating point operations per second for the CPU and GPU. Plot obtained from freely available documents provided by Nvidia.

The advent of multicore and manycore architectures saw them being deployed to speed-up computations across several disciplines and application areas. Prominent examples include semi-numerical algorithms such as sorting [32], graph algorithms [112], image processing [39], scientific computations [46, 27], and also on several irregular algorithms like [139, 108, 123]. Using varied architectures such as multicore processors, the graphics processing units (GPUs) and the IBM Cell researchers have been able to achieve substantial speed-ups compared to corresponding or best-known sequential implementations. In particular, using GPUs for general purpose computations has attracted a lot of attention given that GPUs can deliver 4 TFLOP, or more, of computing power at very low prices and low power consumption.

When using accelerators such as the GPUs and the SPUs in IBM Cell, typically there is a control device that has to explicitly invoke the accelerator. Typically, data and program are transfered to the accelerator, the accelerator computes and returns the results to the device. This approach however has the drawback that the computational resources of the device, a CPU in most cases, are mostly underutilized. As multicore CPUs containing tens of cores are on the horizon, this underutilization of CPUs is a severe wastage of compute power. More recently, in [79], it is also argued that on a broad

class of throughput oriented applications, the GPGPU advantage is only of the order of 3x on average and 15x at best.

In Figure 1.3, we see a typical model of accelerator based computation. In this model all the data and code is sent over to the GPU before any computation begins. After this the accelerator, which is the GPU in this case, does the computation and sends over the results to the host device, which is the CPU. The CPU on the other hand stays idle for all the period of time during which the accelerator is busy.

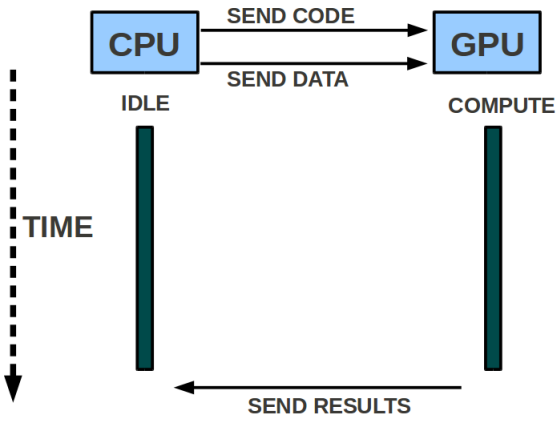

Figure 1.3 Conventional model of accelerator based computing.

1.2.1 Graphics Processing Units (GPU)

We first discuss how accelerator based computing was successful and a paradigm shifting phenomenon. The whole world of parallel computing underwent a change when graphics practitioners found that GPUs can be used for many general purpose computations. The term of General Purpose Computing on Graphics Processing Units (GPGPU) was coined during that time. Although NVidia led the initial development of the GPGPU program, soon significant contributions were made by industry houses and scientists from several fields of research. Today, OpenCL is the dominant open source general purpose computing language for accelerators, although the dominant proprietary framework is still NVidia's Compute Unified Device Architecture (CUDA).

One of the most significant initial results on the GPGPU front was that of the scan primitive proposed by Sengupta et. al [116]. In this work, the authors demonstrated the power of GPU computing by efficiently implementing a highly used computing primitive. Scan is an operation where, we apply an associative operation on a set of contiguous operands. The authors basically employed the `O(logn)` algorithm to compute the scan on an array by performing an up-sweep and a down-sweep. This technique was subsequently aided by applying well known caching and blocking techniques at each level of the computation. The authors demonstrated their implementation on an early GT8800 GPU. They showed that a scan on 1 million elements can be done in less than a millisecond. This was achieved by running 128 threads on each core of the GPU which was previously impossible on any other parallel

architecture. The authors also demonstrated the applicability of their scan primitive by using it in a tri-diagonal solver and a sorting routine.

1.2.2 IBM Cell Broadband Engine

The Cell Broadband Engine (or the Cell BE) [65] is a novel architectural design by Sony, Toshiba, and IBM (STI), primarily targeting high performance multimedia and gaming applications. In Figure 1.4, we see the chip layout of the IBM Cell BE processor. It is a heterogeneous multicore chip that is significantly different from conventional multi-processor or multicore architectures. It consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip.

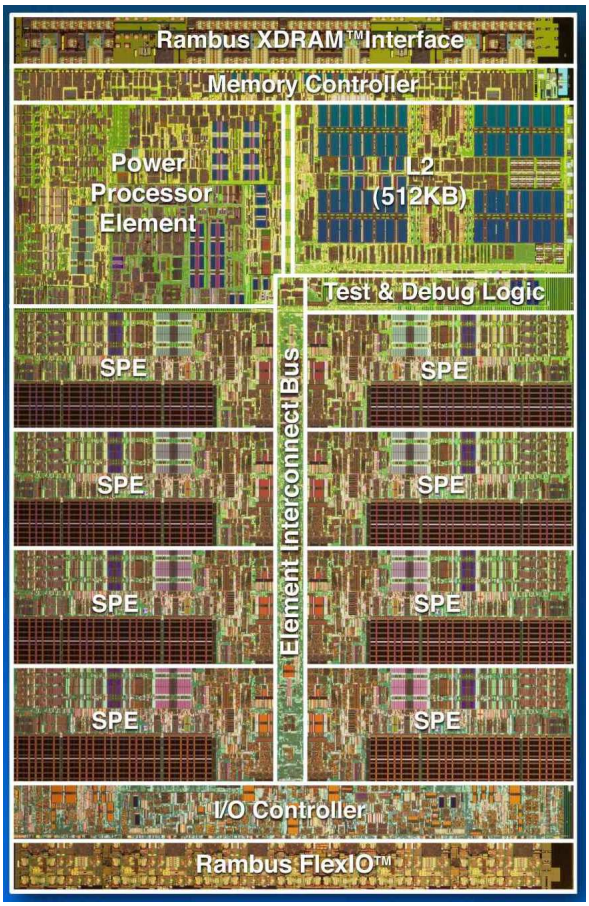

Figure 1.4 The Cell BE Processor. Image obtained from Cell BE Programming Guide distributed by IBM.

The POWER4 processor which was also designed fundamentally for aiding graphics and games, was re-designed to perform general purpose computations and renamed as the Cell. It was heavily used in the Sony Playstation gaming consoles and also contributed towards many petaflop supercomputers such as the IBM Roadrunner. An initial work on the Cell BE was of the implementation of a breadth first search routine [112]. This paper also employed a famous technique known as the Bulk Synchronous Parallel (BSP) model [133] for algorithmically re-designing BFS to suit the Cell BE architecture. The paper employed SIMD operations in each of the Synergistic Processing Elements (SPEs). Proper work distribution from the Power Processing Elements (PPEs) and synchronization among the two led to a 4x performance benefit over the conventional sequential implementations on Pentium machines. The authors demonstrated a performance of around 100 million edges per second.

Another fundamental computing primitive of List Ranking was implemented in the Cell BE architecture by Bader et al. in [11]. Due to the highly irregular nature of list ranking, it is a particularly challenging problem to parallelize on cached based and distributed memory architectures such as the one found in the Cell BE. In this work, the authors describe a generic work-partitioning technique on the Cell to hide memory access latency and apply this to efficiently implement list ranking.

1.2.3 Some other works using accelerators

In a more recent success story, the accelerator based implementations especially on latest generation GPUs also demonstrate the prowess of wide array of throughput cores. Sorting has always been an area of high interest for computer scientists and hence also highly pursued in the domain of massive parallelism. In [81], the authors demonstrated a new randomized sorting technique. In this technique, the authors divided the elements in a highly efficient pattern onto a certain number of bins. This efficient binning helped in a better utilization of the shared resources that are present in a GPU. So, all the bins can be concurrently scheduled onto the several symmetric multiprocessors(SMs) that are available for sorting. The authors showed a performance of around 120 million elements per second on the GTX 280 GPU which was released in 2008. Along the similar lines, there was another comparison sort paper [32] that appeared in 2011 which used the latest Fermi line of GPUs for sorting 32 bit integers. In this paper, the authors showed the optimal usage of not only the shared memory that is available to all the SMs, but also the registers that are available to each of the cores. The major contribution of the paper was the memory access optimizations, the avoidance of conflicts and efficient usage of hardware that is available. The results of the merge sort on GTX 580 was around 140 million elements per second on an input list of 2^{20} elements. The authors also demonstrated the applicability of their technique in variable length keys such as strings. Both of these papers, very efficiently describe the use of massive parallelism that is offered by the accelerators. We learn from these papers that in order to extract the highest possible amount of performance, we need to run as many threads as possible in order to achieve bandwidth saturation, remove irregularities in data access, and maximize shared cache usage.

In Table 1.1, we see a brief of the accelerator results that are currently known to be the state-of-the-art. These results show the workload, the accelerator platform they have been implemented upon and

the current best speed-up percentage they have over conventional parallel implementations on multicore CPU.

Workload	Platform Used	Performance	Speed-Up
List Ranking [139]	GTX 280 GPU	5.7 ms (1 Million elements)	3x
Sorting [81]	GTX 580 GPU	250 MKeys/sec (16 Million elements)	1.7x
SpMV [84]	GTX 280 GPU	21 GFLOPS (2 Million Elements)	5x
Breadth First Search [112]	IBM Cell BE	25 GB/s (2 Million vertices)	5x
Breadth First Search [91]	GTX 580 GPU	3200 MElements/sec (5 Million vertices)	4x
Connected Components [123]	GTX 280 GPU	36 ms (16 Million vertices)	8x

Table 1.1 This table shows the accelerator results over known parallel implementations. Values in the brackets under Performance indicate the input size on which the performance was recorded.

1.3 Beyond Accelerator based computing

Special purpose accelerators are not ubiquitous in nature and hence pushing such accelerators to be so may not be the right choice. Hence, it is required that one make a judicious choice of tasks that one wishes to solve on a given kind of architecture. It is also believed that in future, high performance computing will be dominated by platforms that contain a heterogeneous mix of processors with varying capabilities. In particular Prof. Jack Dongarra quotes [33],

> GPUs have evolved to the point where many real-world applications are easily implemented on them and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs.

In this case we can easily argue the case of many-core vs multi-core computing, where a many-core is a combination of many low compute cores bundled in a chip and a multi-core is a chip where a small number of high-power cores are packed. There are several scientific computations which have high compute requirement, but are aided by graphical simulations which require low compute but a higher number of working threads. Hence, we can intuitively make an assumption regarding the presence of many large scale applications, that are not entirely suitable for accelerator type computations. There has to be some kind of a trade-off that must exist in order to cater to all types of applications. In particular we may look at some cases where there has been sufficient proof to show that some operations like the atomic operations are not very well suited for the accelerators. Computations such as histograms which

are not time consuming ones in the CPU often suffer in GPUs. In paper [113], the authors have shown an implementation of histogram on GPUs where they have dealt with the complications of efficiently computing histograms. The authors have showed the complexity that is involved in the accumulation stage of all the bins and the limited precision which is available. The CPUs, however have proved time and again their efficiency in performing operations such as atomics, gather and scatter. It is the development of the dedicated hardware units for these operations that enable the CPU to excel in their implementations.

In 2010, Intel researchers published a paper which made an attempt to debunk the GPU computing myth [79]. The paper mainly dealt with 14 computing primitives that are commonly used in large applications. The authors implemented each of the workloads on the current generation multi-core CPUs as well as the GPUs. They showed that, with proper optimizations and hardware utilization of the CPU, the GPU results will never out-perform the CPU by the several orders of magnitude as they were happening in many recent works.

The authors primarily revisited some of the works on GPUs that were done previously which claimed improvements ranging from 10x to 100x. However, the authors found that with proper tuning of code on the CPU and the GPU, the gap narrows down to only 2.5x. This is a very significant result that was not expected after the success of the accelerators. The authors mainly focused on workloads which find high application in scientific and financial computations. Workloads like SGEMM, Bilateral, FFT, Convolution, Monte Carlo Simulations and Sorting are widely known and worked upon. In this paper the authors focused on re-designing each of these workloads that will extract the maximum performance out of a multi-core CPU. Techniques such as multi-level caching, data reuse and applying vector SIMD operations as much as possible, were applied. In contrast to many-core designs, these tunings make complete sense as simple assumption that the compiler takes care of all internal optimizations is wrong. Hence, we need to look into the working of each and every instruction on a parallel CPU as we do on an accelerator say GPU. So, in a nutshell, the authors found out an essential truth that accelerators are not the only way ahead and multi-core CPUs can in no way be ignored from the whole picture.

From this work, we eventually can arrive at the idea that what if we are able to put both the multi-core and th many-core processors to work together. A platform where these devices can communicate among themselves was needed to make it possible. With the advent of PCI Express, this problem also has been solved to a large extent as many accelerators and host processors can now communicate among themselves although being bound by the bandwidth.

1.3.1 Non-overlapped hybrid computing

In one direction we can see computations that have two or more different devices working in an interleaved manner. In such case of problems, usually the application is split into heterogeneous work groups which are picked up by a single device based on the compute nature of that particular workload. The typical execution pattern of non-overlapped hybrid execution is shown in Figure 1.5. One of the popular works using this mechanism was proposed by Wei et al. in [139]. In this work, the authors

have demonstrated a split of the input elements using a randomized mechanism. After this splitting, the ranking of each sublist is performed by either the CPU or the GPU in an interleaved fashion. The size of the individual sublists determine the execution platform of that piece. Another interesting work in recent times on hybrid systems was proposed in [58]. In this paper the authors propose a Breadth First Search technique for hybrid platforms. The basic technique applied in this paper is that of architectural optimizations. The authors first present a CPU implementation of graph BFS which is followed by a hybrid implementation. The hybrid implementation is not that of an overlapping nature. The application chooses the CPU and GPU codes optimally depending on the graph type. So, this is more of an interleaved execution rather than the overlapping data parallel execution. The authors, in this case also shows that a fully optimized quad-core performance of a CPU can match that of a GPU in the case of graph traversals which are usually irregular in nature. Results show that the hybrid implementation can process nearly 250 million edges per second using 16 threads on a quad-core CPU. The GPU results are in the range of 300 million edges per second. An interleaved sorting application routine that works on separate work-units applying different algorithms was proposed by Sintorn et al. in [122].

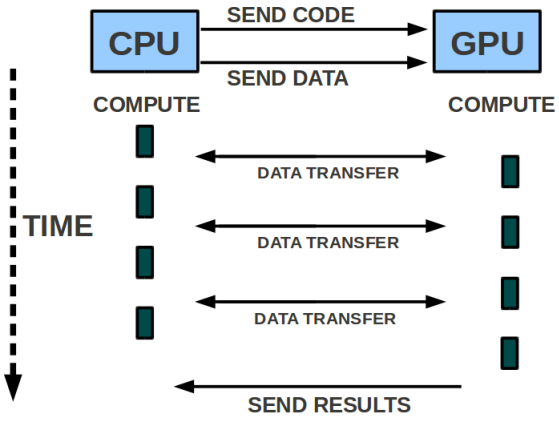

Figure 1.5 Non-overlapped hybrid execution.

1.3.2 Overlapped hybrid computing

The second direction of hybrid computing is that of overlapped computation. A typical overlapped execution pattern is shown in Figure 1.6. We work in most of our problems in this thesis on this track. One of the first works in this direction was proposed by us on solving parallel list ranking in [16]. Further details of this work is provided in Chapter 6. In a more recent work Munguia et al. proposed a hybrid BFS algorithm that works in an overlapped execution pattern in [91]. In this work, the authors have shown a task based partitioning of the work load. This is followed by a work load aware execution phase where work units are allocated to individual devices. The choice of the device for execution is determined by a threshold. A similar work on graph exploration was proposed in [43], where the

authors achieved good performance benefit from using overlapped execution pattern on popular graph exploration algorithms.

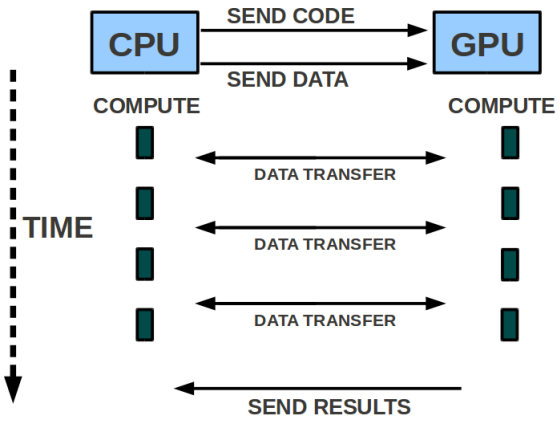

Figure 1.6 Overlapped hybrid execution.

1.3.3 Current Results in Hybrid Computing

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) initiative at the University of Tennessee is one of the earliest works on hybrid computation. The group worked on familiar numerical techniques in hybrid platforms. One of the relevant works from their group is [126]. In this work the authors primarily focus on developing a Dense Linear Algebra (DLA) benchmark suite which would be analogous to the LAPACK benchmark for conventional parallel systems. This DLA suite is proposed purely to be run on hybrid multicore systems. The main motivation of the work stems from the fact that the linear algebra computations are highly compute intensive kernels which involve a lot of double precision operations on real numbers. The solution is based on data parallelism where the CPU acts as the master device and runs the mother thread. Smaller kernels which are inefficient to be run on the CPU and rest are offloaded to the GPU. Also the asynchrony between the two devices are utilized at regular intervals to offload data/code. The authors demonstrate the performance of their solution strategy by designing a hybrid code for LU factorization of matrices. The results are nearly 30% better than the pure GPU implementations.

In Table 1.2, we show the performance and speedups that are obtained as a result of the hybrid implementations over the known accelerator implementations.

Apart from the pure accelerator based computing model such as the ones in [116, 108, 112], hybrid computing can be broadly classified into two generic streams.

Workload	Platform Used	Performance	Speed-Up
List Ranking [16]	GTX 280 GPU + Intel i7 980x	3.6 ms (16 Million elements)	1.5x
Sorting	GTX 580 GPU + Intel i7 980x	290 MKeys/sec (128 Million elements)	2x
SpMV [84]	GTX 280 GPU + Intel i7 980x	9.6 ms (32 Million elements)	1.5x
Breadth First Search[17]	GTX 580 GPU + Intel i7 980x	3500 MElements/sec (16 Million vertices)	1.5x
Connected Components [16]	GTX 280 GPU + Intel i7 980x	21 ms (16 Million vertices)	1.5x

Table 1.2 This table shows the hybrid results and the improvements over the accelerator results. The values in the brackets below the Performance indicates the input sizes on which the performance was recorded. The input element sizes are similar to that used in Table 1.1 for consistancy.

1.4 Summary of this thesis

Accelerator based computing is here to stay, and there has been sufficient evidences to show that heterogeneous computing is better than pure parallel computing in many ways. So, in the near future we will see the shift of pure multicore computing towards *hybrid multicore computing*. In this thesis, our main objective is to study the implications of accelerator based computing and provide parallel primitives for the hybrid platform. Some of the questions we will be finding answers include: how to arrive at efficient algorithms in the hybrid computing model for fundamental problems in parallel computing such as list ranking, sorting, and how to analyze a hybrid algorithm for its efficiency, what are the parameters that are important for a hybrid algorithm, and the like. In addition, we will focus on optimization and tuning of hybrid systems. In our work we will also propose a characterization of the hybrid workloads and how the implementations can be made platform aware.

The main goal in this thesis work would be to understand hybrid computing solutions in general. This would include proposing newer models of computations, design of algorithms, and arriving at guidelines for developing general purpose hybrid solutions. Issues like optimum resource utilization, power management, and speeding up known parallel applications would be undertaken. The thesis therefore argues that future parallel computing solutions would necessarily have to include elements of hybrid multicore computing so as to be fast, efficient, and scalable.

In this thesis we broadly explore two different categories of workloads. In Part I of the thesis, we focus on some semi-numerical computations like that of sorting and pseudo randomness. We work in both task and data parallelism in these problems to achieve optimal results. In Part II of the thesis, we concentrate on some graph algorithms which are mostly irregular computations. We attempt to formulate ideal solutions for graph problems like list ranking, connected components and breadth first search.

Chapter 2

GPU and CPU Parallelism

2.1 Introduction

The microprocessors that are based on the single central processing unit (CPU) such as the ones that are in the Intel Pentium series processors or the AMD Opteron family, encouraged a massive performance increase and cost reductions in majority of the massively used computer applications. These units brought Giga (billion) floating point operations per second (GFLOPS) to the desktop and hundreds of GFLOPS to clusters. This gain in performances have allowed the software applications to provide more functionality and generate more useful results. The users have demanded more improvements once they have got accustomed to these improvements which has resulted in the creation of a paradigm shifting cycle in the computer industry.

As a result of these improvements, the developers have mostly relied on the advancements made in the hardware industry. The same applications simply run faster with every new generation of processors that are introduced. Since 2003 however, this drive has dissipated owing to the energy consumption and heat dissipation issues. The processors have been limited in their clock frequency, and the level of productive activities that can be performed in each clock cycle. A majority of microprocessor vendors have shifted to multiple processing units or *processor cores*, that are used in a single chip to increase the processing power. The shift has made a tremendous impact in the software community [124].

Traditionally, a majority of the software applications were written as sequential programs that is described by von Neumann [138], in his seminal work. The execution of these programs can be understood as a human stepping sequentially through the lines of instructions. These programs are naturally expected to run faster with each new generation of processors. However, such exception is no longer valid. A sequential program will run on only one of the processing cores which will not be significantly faster with each generation. Without the use of performance engineering, applications will not be able to accommodate new features and capabilities with each new generation of processor. This will hence reduce the growth opportunities in the entire computer industry. Application softwares that will continue to enjoy performance improvements with each new generation of processors will be *parallel programs*.

The new incentive for parallel program development has been termed as the "concurrency revolution" [124]. However, the practice of parallel programming is in no means new. The community of high performance computing has been existing for decades. The programs written mostly ran on large scale expensive computers. In todays world when all the commodity microprocessors available to the users are parallel computers, the application development drive for parallel programs have risen dramatically.

2.2 Multicore and Manycore

Since the revolution of parallel computing in 2003, the semiconductor industry have settled on two main trajectories for designing microprocessors [62]. The *multicore* trajectory attempts to maintain the execution speed of sequential programs while transitioning into multiple cores. The multicore began as two-core processors which continued to double the number of cores with each new generation. The current example being the Intel i7 processor family. It has 8 processing cores, each of which is an out-of-order multiple instruction issue processor implementing full x86 instruction set. The microprocessor supports simultaneous multi threading (SMT) or hyperthreading with two hardware threads and is designed to maximize the execution speed of of sequential programs.

The other trajectory has been that of *many-core*. The many-cores focus more on execution throughput of applications. The many-cores began as a large number of much smaller cores which also kept doubling with each new generation. The current example being the NVidia Kepler GTX 680 graphics processing unit (GPU) and has 2496 cores. Each of these cores too are heavily multi-threaded, in-order, single instruction issue processor that shares the control and instruction cache. The GPUs have predominantly led the race of GFLOPS since 2003. The time line of the development has been shown in Figure 2.1. While the performance improvement of general-purpose microprocessors have slowed significantly, the GPUs have continued to improve relentlessly. The processing figures shown in Figure 2.1, does not necessarily mean the application speeds but are the theoretical processing peaks that can be achieved.

At this point, there might be an obvious question regarding the performance gap between the CPU and the GPU. The reason lies in the basic design philosophies that exist between the CPU and the GPU. The CPU is designed for sequential code performance. There is a greater level of sophistication in the control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, the large cache memories that are provided to reduce the instruction and data access latencies of large complex applications.

The design philosophy of the GPUs is shaped by the fast growing video and gaming industry. This industry demands exceedingly high performance which exerts a massive economic pressure to perform higher GFLOPS per video frame in high-end gaming applications. This demand makes the GPU scientists to maximize the chip area and power budget dedicated to floating point operations. The most heavily used solution is to employ massive number of threads. The hardware takes advantage of the

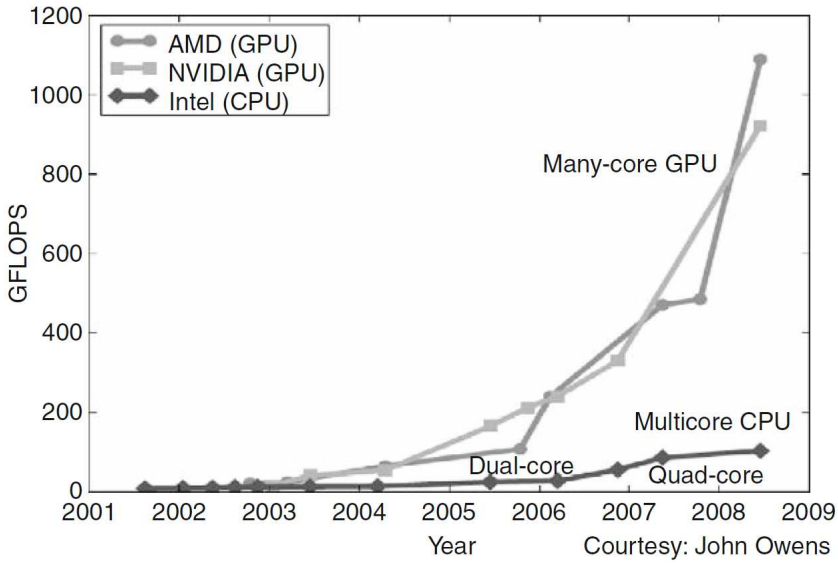

Figure 2.1 Performance gap between CPUs and GPUs.Image obtained from [71].

large number of execution threads to find work to do when some of them are waiting for long latency memory accesses. This minimizes the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so that all the threads do not need to access the DRAM. As a result more chip area becomes available for floating point calculations.

2.3 GPU Architecture

The Graphics Processing Units (GPU) were initially designed with fixed units which were not programmable. As the transistor budgets went up and flexibility rose, each of the hardware components were made programmable. Although current generation GPUs still hold some specific hardware components, they are mostly considered to be generic compute accelerators.

The GPUs are generally organized as an array of highly threaded streaming multiprocessors (SM). The number of SMs that are present in the GPUs vary from generation to generation. Each of the SMs have a number of cores or streaming processors (SPs) that share common control logic and instruction cache. Each of the GPUs have multiple Graphic Double Data Rate (GDDR) DRAM which is commonly referred as the global memory. These GDDR DRAMs differ from the DRAMs that are found on CPU motherboards in the frame buffer memory that is used for graphics purposes. For common graphics applications, they hold video images and texture information for 3D rendering. For compute, they function as a very high bandwidth off-chip memory. For massively parallel applications the higher bandwidth that is offered makes up for the longer latency of the global memory.

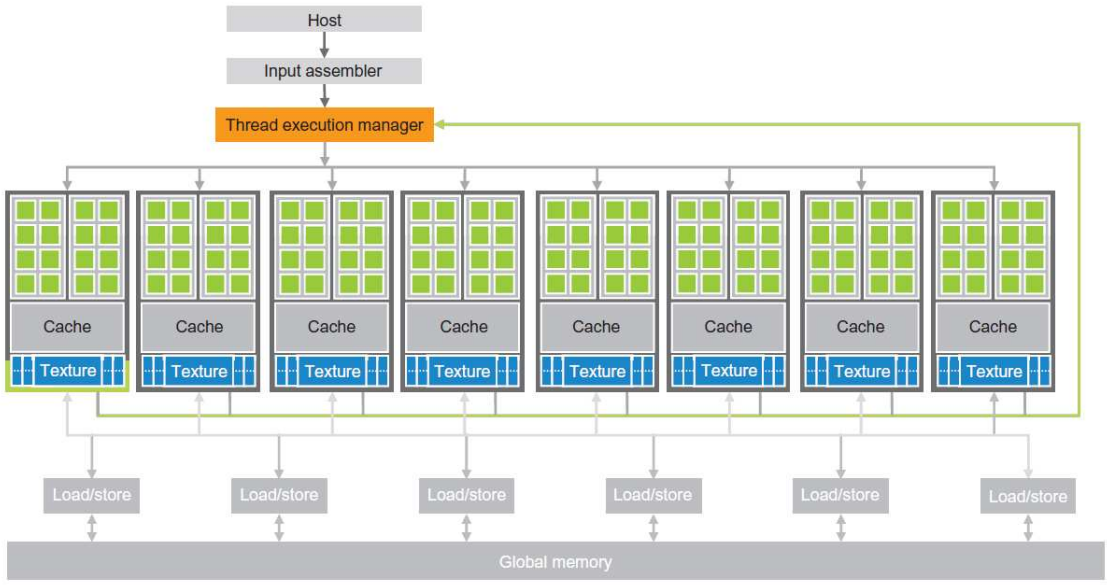

Figure 2.2 Basic GPU Architecture.

Figure 2.3 A Nvidia Tesla 4 GPU, Nvidia GTX 580 GPU and a GTX 600 Series GPU. The images are obtained from freely available documents provided by Nvidia.

The basic GPU architecture is shown in Figure 2.2. It is an abstraction of the generic GPU architecture that is put out by Nvidia. Figure 2.3 shows the Nvidia GPUs that are commercially available in the market as off the shelf devices. These GPUs are commonly used for either desktop computing or for building large cluster servers.

A current generation Fermi GPU has a theoretical peak of 1.5 TFLOPS. Each of the multiprocessors or the SMs execute in Single Instruction Multiple Data (SIMD) mode. That is, each of the thread processor in a SM executes the same instruction simultaneously. The SIMD width of the current generation GPUs is 8. Nvidia typically defines this execution model as Single Instruction Multiple Thread (SIMT).

Each core of the GPU is clocked at 1.5 GHz and a GTX 580 Fermi GPU has 16 SMs in one card. So the total number of cores that is present in the GPU if there are 32 SPs or cores is $16 * 32$ or 512.

Each of the multiprocessors execute in an asynchronous fashion. There is no communication across the multiprocessors although there are synchronization mechanisms for groups of threads that are running inside each SM. A group of threads that are scheduled into any SM at a particular time can always communicate amongst themselves. The threads of each SM can also communicate via the shared memory that is present in each SM. The communication mechanism for the multiprocessors is via the high bandwidth off chip global memory. The operations on the common global memory locations by multiple processors do not follow any order, making it hard to communicate. Only one of the multiple simultaneous operations on the memory succeeds, making it non-deterministic in nature. Atomic operations on the hardware are meant to overcome above problem; they guarantee all operations to succeed but no information on ordering of the operations is known.

Each multiprocessor has a special function unit which performs operations like, divide, square root, etc. It is slower compared to other processing units but is infrequently used. Each multiprocessor has a high bandwidth, low latency on-chip local memory (shared memory). Shared memory is a valuable resource but is limited to 16KB on current generation GPUs. There is also a high-bandwidth, high-latency large off-chip global device memory, over 1GB on high-end models. Device memory needs to be filled via the PCIe bus by the host from the host memory. Data on the GPU device memory can be pulled back to the host memory in a similar way. Recent software improvements have enabled the use of host memory from the device, thus extending the amount data which can be processed beyond GPU device memory.

Eight thread processors are grouped together to form a multiprocessor and several multiprocessors are combined to form a device in the hardware model. Threads are combined together as thread blocks in order to group large number of threads. These thread blocks form a grid of blocks, which are processed on the GPU using a kernel. Each thread block is executed on a multiprocessor and thread of the block can communicate through the shared memory of the multiprocessor. Synchronization can also be triggered for the threads of a block using barriers. More than one thread block can occupy a multiprocessor. These thread blocks are time-shared by interleaving warps of threads. No ordering of the threads warps is guaranteed. The number of thread blocks which can be handled by a multiprocessor depends on the resources used by each thread block. Private partitions in the shared memory and register file are logically created for each thread block that occupies a multiprocessor.

A large number of threads is required for extracting enough parallelism on the GPU. The SIMD instructions being executed on each multiprocessor may stall the processors if the instruction takes a long time. A memory request from the global memory may take as much as 500 clock cycles. Time sharing a large number of threads on a multiprocessor can improve the overall throughput of instructions. Thread warps which are context switched can belong to the same block or different blocks being executed on a multiprocessor. GPUs are also designed for structured data accesses, as they are designed for graphics

processing. The memory is efficient when transactions are performed in an ordered manner by the threads.

The current generation GPUs coalesce as long as all the reads/writes are from a block of memory of 64 or 128 bytes. Shared memory has a low latency but can suffer from bank conflicts. Shared memory of 16KB is divided into 16 banks of 1KB each. Each consecutive word of the memory is placed in consecutive banks, making it possible to read 16 words by a single half-warp (16 threads) without any bank conflicts. In case of a bank conflict, the requests are serialized.

Above resources and limitations should be kept in mind for an efficient implementation on the GPU. Data common to a thread block and required for processing more than once should be stored in the shared memory. The life of the data in shared memory is that of the thread block and that data can not be referred by other thread blocks. Data transaction from the global memory should be coalesced in nature to achieve many fold performance as compared to non-coalesced reads and writes.

Each multiprocessor is equipped with a thread control unit (Figure 2.2) which manages the scheduling of threads from multiple blocks assigned to the multiprocessor. All 8 thread processors perform the same instruction in a clock cycle. Multiprocessors also have the synchronization units which allow synchronization of threads from a block. The most common use of synchronization is to maintain data consistency when multiple threads are used to read data from global memory to shared memory, assuming the data brought in can be used by other threads of the block.

2.4 GPGPU

Programmability of the GPUs has grown over the last decade. Shader model 3.0 introduced fully programmable vertex and pixel processing units. Support for vector operations on IEEE single precision floating point numbers was introduced. Many high level languages like Cg by Nvidia, open standard GLSL, HLSL by Microsoft etc. for shader programming made it easier to program the GPU and access the computation power. It was studied that GPUs could accelerate some problems by an order of magnitude over the CPU. With the introduction of shader model 4.0, an additional geometry generation unit was added to the pipeline (Figure 2.4). General-purpose application on the graphics processing unit (GPGPU) were mostly addressed through the pixel shader unit and was neither affected nor gained much with the new shader model.

The GPGPU approach could address various non-graphics problems like in-game simulation of physics and computational science. Given the earlier development on the GPUs was focused on graphics applications, the programming environment was tightly constrained. Lack of exposure to the underlying architecture also made it hard for non-graphics developer to port their applications to the GPU. The developer was expected to be an expert in computer graphics in order to make effective use of the GPU.

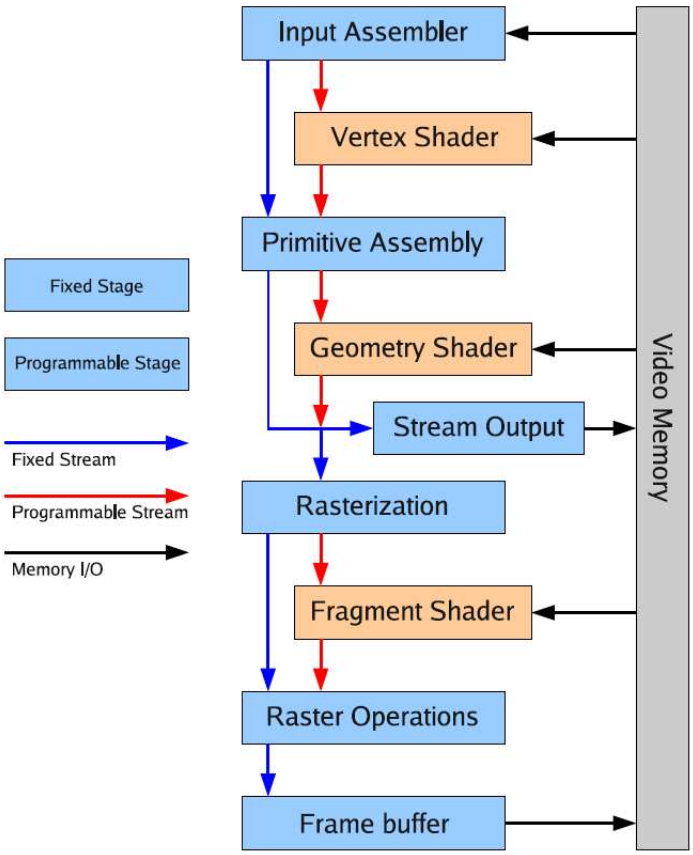

Figure 2.4 Shader Model 4.0 Pipeline. Geometry shader was introduced as new programmable unit in the pipeline.

General purpose computation used a Stream Processing model where a series of operations (kernel functions) are applied to each element from the set of data (a stream). A typical GPGPU problem is mapped as a texture manipulation problem using the graphics pipeline. The main source of input and output data containers are textures which earlier had access only from pixel shaders but now can also be accessed from vertex and geometry shader (Figure 2.4) due to a unified shader model introduced with shader model 4.0. The major limitation of this model was the limited scope of writing an output since scatter was not supported. This approach had a significant learning curve but yet provided opportunities for extreme speedups for selected applications.

2.5 Compute Unified Device Architecture (CUDA)

CUDA is a programming interface to the parallel architecture of the GPU for general purpose computing. This interface is a set of library functions which is coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-

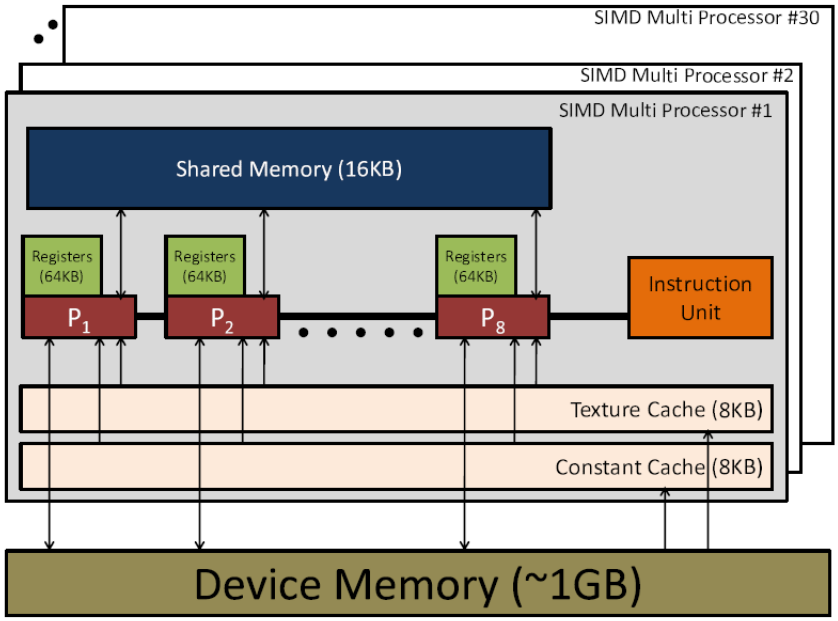

Figure 2.5 CUDA Hardware Model.

core co-processor. The CUDA design does not have memory restrictions of GPGPU. One can access all memory available on the device with no restriction on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the GPU for general purpose computing.

CUDA Hardware Model: At the hardware level the GTX 580 processor is a collection of 16 multiprocessors, with 32 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. In any cycle, each processor of the multiprocessor executes the same instruction on different data. Communication between multiprocessors is through the device memory, which is available to all processors of the multiprocessors. A diagrammatic representation of the CUDA hardware model is shown in Figure 2.5.

CUDA Programming Model: We show the organization of the CUDA software model in Figure 2.6. For the programmer, the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU, 32 on present GPUs. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) is mapped to a multiprocessor at a given time. Multiple blocks can be assigned to a multiprocessor and their execution is time-shared. A single computation on a device generates a number of blocks. A collection of all blocks in a single computation is called a grid. All threads of the blocks mapped to a multiprocessor divide its resources equally amongst themselves. Each

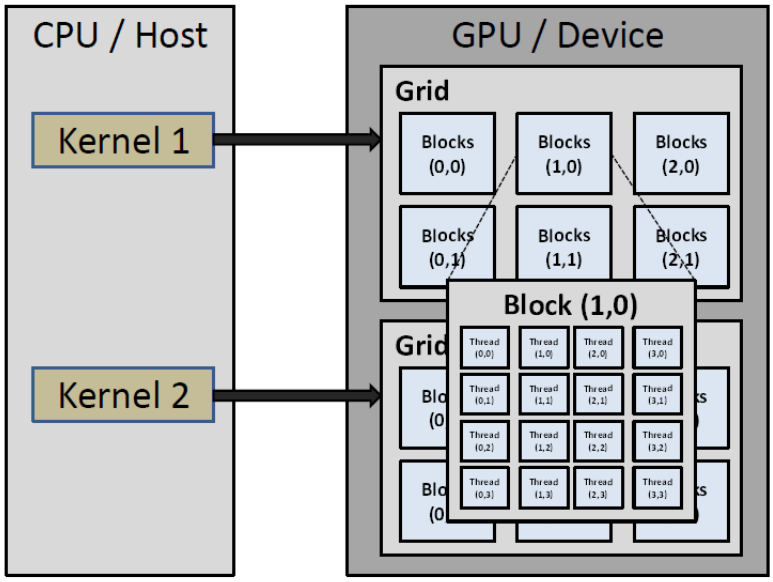

Figure 2.6 CUDA Software Model.

thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. GPU is a co-processor to the CPU and needs to be initiated by the CPU. A typical CPU/GPU application is executed in the following order when initiated by the CPU.

1. Copy data from main memory to GPU memory
2. CPU instructs the process for GPU execution
3. GPU executes the program in parallel using many cores
4. Copy the results from GPU memory to main memory

GPGPU approach using the graphics pipeline had a steep learning curve due to unfamiliarity of programmers with the graphics APIs. CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs.

1. Scattered reads/writes: Code can read from arbitrary addresses in memory.
2. Shared memory: CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher band- width than is possible using texture lookups
3. Faster downloads and read-backs to and from the GPU
4. Full support for integer and bitwise operations, including integer texture lookups.

A sample CUDA code for vector addition looks as follows:

```
__global__ void vecAddKernel(float *A, float *B, float *C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    if(i < n)
      C[i]=A[i]+B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, h_C
    // I/O to read h_A and h_B elements

    dim gridA(512, 512);
    threads=512;

    vecAddKernel<<<gridA, threads>>>(h_A, h_B, h_C);
}
```

2.6 Multicore CPU

CPUs have undergone a massive change in their conventional architecture from the time of single core high clock frequency generations to the current multi-core CPUs. This change was primarily fueled by the conventional architectures hitting the power wall and the Instruction Level Parallelism (ILP) wall. The commodity CPUs available today are commonly called multi-core processors because of the high compute power available in each of the cores which is a distinct difference with the many-core GPUs.

An important component in the basic architecture of a multicore processor is the cache structure. The cache is basically a high speed buffer memory that is provided on the same die as the microprocessors in order to facilitate faster access based on spatial or temporal locality. In the earlier days, there were only a single level of cache that was provided, and each cache miss involved a corresponding penalty in memory access. In the current generation of multicore processors usually a more complicated cache structure can be observed. Current generation of processors typically have two levels of cache that is L1

and L2 caches on the chip of the microprocessor. It was observed, that after the creation of these two levels of cache, the performance of each of the cores could be enhanced significantly. However, with more real life problems coming inside the realm of parallel computing, communication and synchronization among the different cores became a major issue. It is therefore that the architects added a thord level of off-chip cache which is commonly referred to as the L3 cache and is significantly higher in size than the L1 and L2 caches. Data transfer among the different levels are still done in terms of bolcks (cache lines) and conventional coherence protocols are used. The multicore processor which we use for most part of our implementations is the Intel i7 980x which is based on the Westmere microarchitecture from Intel. It has 64 KB per core L1 cache, 256 KB per core L2 cache and 8 MB of shared L3 cache.

From the point of view of performance, the broad optimization strategies that are needed for the CPUs are approximately same as that of the GPUs. The Intel processors can be easily programmed using the OpenMP specifications for creation and management of the threads. The CPU also has got dedicated units for thread management and hence incurs very less overhead. Also, it is important that for any hybrid implementation, we carefully optimize the implementation of the CPU side of the code. The Intel architectures offer the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) for intrinsically executing the SIMD code across the available resources on each of the cores. Each of the cores as already stated has a local cache and a shared cache. It is important to take care about maximum data reuse via the efficient use of each of these shared caches. Apart from that the 128 bit SSE registers available are also needed to be efficiently used so that the code does not become instruction bound. In all the hybrid implementations it is hence of high importance to maintain a parity between the optimization strategies that is employed for the GPU as well as the CPU. This makes comparison of performance and overall gains more intuitive. A sample AVX code to load and set the memory location is shown below :

```
 #include <stdio.h>
 #include <stdlib.h>

// Required for AVX Instructions
 #include <immintrin.h>

int main( int argc, char *argv[] )
{
    float a = 6.0f;
    float b = 19.0f;

    // Following are using 128 bit SSE registers
```

```
    // Set SSE1 register to zero
    __m128 SSE1 = _mm_setzero_ps();
    // Load value of a into SSE2 register
    __m128 SSE2 = _mm_set_ps1(a);
   // Load value of b into SSE3 register
    __m128 SSE3 = _mm_set_ps1(b);
   // Adds value of a and b
    __m128 SSE4 = _mm_add_ps(SSE2, SSE3);

    // Following are doing the same thing as
       above using 256 AVX registers

    __m256 AVX1 = _mm256_setzero_ps();
    __m256 AVX2 = _mm256_set1_ps(a);
    __m256 AVX3 = _mm256_set1_ps(b);
    __m256 AVX4 = _mm256_add_ps(AVX1, AVX2);

    // Sets the 128 bit register to zero
    __m64 MMX1 = _mm_setzero_si64();
    // Sets the two 32 bit integers in reverse order
    __m64 MMX2 = _mm_setr_pi32(6, 6);
    __m64 MMX3 = _mm_setr_pi32(19, 19);
    // Adds MMX3 value to that of MMX2
    __m64 MMX4 = _mm_add_pi32(MMX2, MMX3);

    return 0;
}
```

2.7 Open Multi-Processing (OpenMP)

The last decade has seen a tremendous increase in the widespread availability and affordability of shared memory parallel systems. Not only have such multiprocessor systems become more prevalent, they also contain increasing numbers of processors. Meanwhile, most of the high-level, portable and/or standard parallel programming models are designed for distributed memory systems. This has resulted in a serious disconnect between the state of the hardware and the software APIs to support them. The goal of OpenMP is to provide a standard and portable API for writing shared memory parallel programs.

Over the last several years, there has been a surge in both the quantity and scalability of shared memory computer platforms [24]. Quantity is being driven very quickly in the low-end market by the rapidly growing PC-based multiprocessor server/workstation market. The first such systems contained only two processors, but this has quickly evolved to four- and eight-processor systems, and scalability shows no signs of slowing. On the software front, the various manufacturers of shared memory parallel systems have supported different levels of shared memory programming functionality in proprietary compiler and library products. In addition, implementations of distributed memory programming APIs like MPI are also available for most shared memory multiprocessors. Application portability between different systems is extremely important to software developers.This desire, combined with the lack of a standard shared memory parallel API, has led most application developers to use the message passing models. This has been true even if the target computer systems for their applications are all shared memory in nature. A basic goal of OpenMP, therefore, is to provide a portable standard parallel API specifically for programming shared memory multiprocessors. There are other implementation models that one could use instead of OpenMP, including Pthreads [94], MPI [100], HPF [74], and so on. The choice of an implementation model is largely determined by the type of computer architecture targeted for the application and the nature of the application.

OpenMP is primarily designed for shared memory multiprocessors. The important aspect for our current purposes is that all of the processors are able to directly access all of the memory in the machine, through a logically direct connection. Details on how a machine provides the programmer with this logical view of a globally addressable memory are unimportant for our purposes at this time, and we describe all such systems simply as shared memory.

OpenMP is not a new computer language; rather, it works in conjunction with either standard Fortran or C/C++. It is comprised of a set of compiler directives that describe the parallelism in the source code, along with a supporting library of subroutines available to applications. Collectively, these directives and library routines are formally described by the application programming interface (API) now known as OpenMP. The official OpenMP 4.0 documentation can be found at [1].

A sample OpenMP code to add an array of numbers looks as follows:

Code:

```
int main()
{
```

```
    int i;
    int tot=0;
    int a[12]={12,23,11,3,7,84,23,45,59,20,4,1};

    omp_set_num_threads(4);

    #pragma omp parallel for private(i) shared(tot) schedule(dynamic,3)
    for(i=0; i<10; i++)
      tot+=a[i];

    printf("Total is : %d\n",tot)
}
```

Output:

```
Total is : 291
```

There are certain critical things that are to be noticed from the OpenMP code above. In the first place, we see the #pragma compiler directive that is used to perform an auto parallization of the addition code. In this style of parallization, we typically ask the compiler to perform the parallization as the operation is fairly data parallel. In order to maintain consistancy of the results produced, it is important to point out the variables that are private to each thread and the ones that are shared. In the code above, we see that the loop iterator i has been declared as private as, the iterator should not be risked by any thread to be modified by some other thread. The tot variable which is accumulating the sum, is declared as a shared variable, as it is going to be used by each thread. Hence, if it is not shared, there would be practically no summation and each thread will compute some partial sums.

The second important faceat of the above code is the schedule. We mention the schedule of the above addition as dynamic as we intend to make the scheduling as much dynamic as possible. The other alternative is a static schedule. In a static schedule, the chunk size (the second parameter in the schedule) is compulsorily computed upon by each thread. Hence, it may happen that in some other application with more heterogeneous workloads, each thread will wait until the work on its chunk is completed before it proceeds with the next chunk. This creates an unnecessary wait time. Dynamic schedule on the other hand is more flexible where the chunk sizes are allocated dynamically. So, whenever a thread is finished with its own set of work, it can be immidiately assigned another chunk without wasting any time. However, in many applications, a static schedule is important for maintaining proper data dependencies and meeting synchronization barriers.

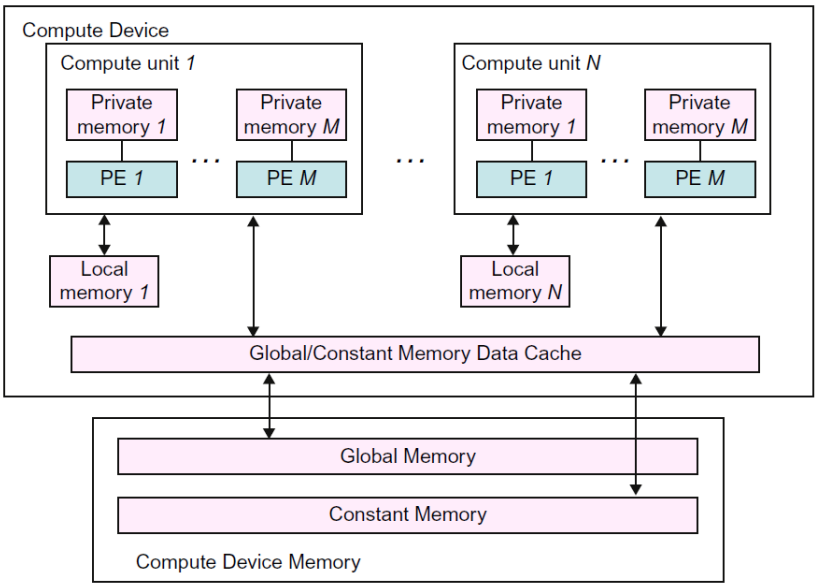

Figure 2.7 OpenCL device architecture. Image obtained from OpenCL Specification.

2.8 Open Computing Language (OpenCL)

OpenCL is a standardized cross-platform programming language based on C. The design of OpenCL was necessitated by the fact that there was an urgent need for the development of portable parallel programs which has heterogeneous computing devices. There was a need for a standardized high-performance application development platform for a variety of fast growing computing devices. CPU based computing has mostly depended on standards such as OpenMP which usually do not encompass SIMD functionalities. For heterogeneous computations, CUDA usually has constructs for addressing memory hierarchies and SIMD executions but has been mostly device specific. These limitations make it difficult for an application developer to access the computing power of CPUs, GPUs, and other types of processing units from a single multi platform source code base.

The development of OpenCL was initiated by Apple Inc. and is maintained by the Khronos group. OpenCL heavily draws inspiration from CUDA in the areas of supporting single code base for heterogeneous platforms, data parallelism and memory hierarchies.

OpenCL uses a data-parallel model for execution that has direct correspondences with CUDA. An OpenCL program typically has two parts, the kernel which runs on one or more parallel devices and a host program that controls the kernels. The typical model is to write the host program to launch several kernels for parallel execution. When a kernel is launched, its code is executed by *work-items* which is the equivalent of CUDA threads. There is an index space that defines the mapping of data to the work items. Work items collectively forms the *work-groups* that corresponds to CUDA thread blocks. Work-items in the same work-group can synchronize among themselves using barriers such as `__syncthreads()` in CUDA.

OpenCL models a heterogeneous parallel computing system as a host and one or more OpenCL devices. In Figure 2.7, we can see a conceptual model of the OpenCL device architecture. Each device consist of one or more Compute Units (CUs) which corresponds to CUDA SMs. A CU can also correspond to a CPU core or other types of execution units like FPGAs or DSPs. Each CU in turn has one or more processing elements (PEs) which corresponds to CUDA SPs. Computation on a device ultimately happens on a PE. Like CUDA, OpenCL too exposes a hierarchy of memory systems that can be used by programmers. These memory types are typically global, constant, local and private. Unlike CUDA, the constant memory can be dynamically allocated by the host. Like CUDA, the constant memory supports read/write access by the host and read-only access by devices. To support multiple platforms, OpenCL provides a device query that returns the constant memory size supported by the device.

2.9 Hybrid Platform

We use two different hybrid platforms for conducting the experiments. One is the high end platform which is typically the one that is used for developmental purposes and is composed of server class hardware. The other platform is the one that is commonly used in the commodity desktop and laptops. We specifically choose to use these two platforms in order to show the advantage of hybrid computing in a wider spectrum of high performance computing research.

2.9.1 Hybrid High

Our high-end hybrid platform is a coupling of the two devices described above, the Intel i7 980 and the Nvidia GTX 580 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1. The CUDA API Version 4.1 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution.

In this thesis work, we mainly use the GTX 580 GPU from NVidia. GTX 580 is a current generation Fermi micro-architecture from NVidia and has 16 streaming multi-processors (SM). Each SM has 32 cores which gives us a total of 512 compute cores which are each clocked at 1.54 GHz. Each of the SMs of the latest Fermi processors have a hardware scheduler which schedules 32 threads at a time. This group is called the warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU now has a fully cached memory access via an L2 cache, 768 KB in size. In all, the GTX 580 has a peak single precision performance of 1.5 TFLOPS.

Along with the GTX 580, we use an Intel i7 980x processor as the host device. The 980x is based on the Intel Westmere micro-architecture. This processor from the Intel family with each core running at 3.4 GHz and with a thermal design power of 130 W. The i7-980X has six cores and with active SMT(hyper-threading) can handle twelve logical threads. The L3 cache has a size of 12 MB. The L1

cache size is 64 KB per core and L2 is 256 KB. Other features of the Core i7 980 include a 32 KB instruction and a 32 KB data L1 cache per core and the L3 cache is shared by all 6 cores.

We mostly use the above mentioned platform for our experiments. Apart from that we also include some results using the NVidia K20 processor which is the latest offering from NVidia. The NVidia K20 is the first processor in the Kepler series which was released in 2012. The K20 processor offers dynamic parallelism with both L1 and L2 caches. The K20c processor that we use has 13 SMs with 192 CUDA capable cores on each of the SMs making a total of 2496 cores. Each of the cores are clocked at 2600 MHz and has a global memory of 5 GB. The L2 cache size is 1.25 MB and L1 cache size of 48 KB. The K20c is also coupled with a Intel i7 980x processor and makes our third platform for experimentation.

2.9.2 Hybrid Low

Our low-end hybrid platform resembles a commodity desktop computing environment more closely. The Hybrid-Low platform is a combination of an Intel Core 2 Duo E7400 CPU along with an NVidia GT520 GPU.

The Intel Core 2 Duo CPU is one of the earliest multicore offerings from Intel and was released in the year 2008. It has 2 cores with hyper-threading and each of the cores are clocked at 2.8 GHz. The CPU consists of a 3 MB L2 cache and the maximum power consumption is around 65 W. The CPU was designed entirely for commodity PCs which gives a sustained performance of about 20 GFLOPS.

The GT520 is a stand-alone graphics processor having 48 computing cores and 1 GB of global memory. Each of the compute cores are clocked at 810 MHz. The GPU on an average give a sustained performance of 77.7 GFLOPS and consume about 29W of power. In this system both the processors are of a comparable performance range and hence provide a more realistic platform for experimenting the hybrid programs.

Chapter 3

Our Contributions

In this chapter we discuss the main ideas and contributions that we are presenting in this thesis. We first discuss the fundamental problems and questions that we are trying to answer and then showcase the results that we have obtained.

3.1 Motivation

An accelerator-based computing model typically involves adding an accelerator device to a host. The accelerator is attached to the host via a PCI Express link, or could be sharing the same board with the CPU. GPGPU, the acronym for general purpose computing on GPUs, follows the former model.

Most of the current literature on GPGPU considers the CPU as a host device and pushes most of the computation to a GPU [116, 139, 108]. The CPU is practically idle in the computation process. As issues such as performance and power dominate the present generation solutions for high performance computing, the current practice of not utilizing a portion of computing resources is hardly the way forward. Some of the advantages that are possible due to a combined CPU + GPU hybrid computing model are:

- **Performance Efficiency:** It is the case that one particular processor may not be best suited for all operations. Hence, in a given computation, it is likely that parts of the computation which are expensive on the GPU are better performed on the CPU and vice-versa. So offloading the expensive operations of one device onto another is a logical choice to improve performance efficiency.

- **More Data and Task Parallelism:** One can extract more data parallelism from the hybrid model in which we can make all the computational devices work in unison. This is unlike the usual case where one processor performs a certain computation and the other is idle. A higher amount of task parallelism is also possible as data parallelism is mostly exploited through CPU vectorization and CPU warp level SIMD executions. In most of the applications analyzed in this thesis, we have tried to design the algorithm in which it would be possible to allocate heterogeneous work units

to each device which can then be executed in parallel to each other in an interleaved/overlapped manner.

- **Functional/Pipelined Parallelism:** One can benefit via functional or pipelined parallelism also, where different functions are processed on different devices.

- **Programming Productivity:** New programming environments like OpenCL are helping developers to write hybrid multi-core programs in a seamless manner.

Hence, it is important to employ a hybrid model which will utilize both the devices for solving a particular problem. We can call an algorithm as a *hybrid multicore algorithm* if it is designed to run on a hybrid computing platform. Hybrid algorithms are gaining attention recently [140, 126, 50]. However, using a collection of heterogeneous processors is highly non-trivial as it involves issues such as the availability of a suitable programming model, synchronization, and communication mechanisms, among others.

Fortunately, the programming support from the several vendors presently allows one to write multi-architecture programs. The current GPU models, and other leading GPU platforms, allows kernel calls to be executed in a non-blocking manner[29]. This is termed as *asynchronous concurrent execution*. It means that the CPU, which initiates the kernel call can execute other CPU instructions while the kernel is under execution on the GPU.

While the benefits of hybrid algorithms, and their programmability are clear, there are several analytical questions that have to be answered to arrive at efficient hybrid algorithms. For instance, it is likely that transfer of intermediate results between the CPU and the GPU may introduce certain delays at either end. These delays can mean that either the CPU, or the GPU, or both, may be idle during certain time periods. For an efficient hybrid multicore algorithm, one should minimize these idle times. It may also be important to see which idle time can be tolerable. For example, keeping the GPU idle for one second may mean a loss of more FLOPS than keeping the CPU idle for the same amount of time. Thus, a hybrid algorithms has to make the right choices in its execution plan.

In a similar fashion, when hybrid algorithms are designed in a functional/pipelined parallel setting, the goal should be to assign the right task on the right processor. In this case, it is not clear at the outset, how to arrive at this assignment so as to minimize the total execution time and the total idle time, among possibly other things.

3.2 Key Targets

Through this thesis work, we attempt to establish hybrid multicore computing as a basic parallel computing model by (re)designing parallel algorithms for some basic computer science problems. We implement these algorithms on different platforms consisting of modern generation multicore and many-core architectures and validate our results.

In the recent past there have been many works that have shown the massive advantages of commodity parallel processors such as Graphics Processing Units (GPU) as well as multi-core CPUs. From the point of heterogeneous parallel computing, we need to think about the following fundamental motives while solving our problem.

1. **Work Allocation :** How to allocate the right job to the right processor ? It is a known fact that, all the parallel processors that are commonly available today are not ideal. There has been no specific work to show that a particular processor is right for all types of computations. So, when we have access to multiple parallel processors on a particular platform, how do we decide on allocating the right job to the right processor.

2. **Load Balancing:** Load balancing as always, is a fundamental challenge in any distributed or parallel application. We need to decide on strategies that would optimally partition work amongst the available processors without incurring a massive overhead.

3. **Communication:** The connectivity that exists among the processors are still in a naive state. There is no high-speed link that is available in a tightly coupled system that has a very low latency of communication. Hence, it becomes important to address this issue where we propose use of data structures and algorithms that minimizes communication costs between the processors.

4. **Synchronization:** The parallel processors that are available will work in a concurrent fashion during the execution of an application. These processors needs to talk among themselves in order to produce correct results. However, due to the unavailability of any high speed link between the processors, the synchronizing constructs can produce heavy overheads. We address this issue in our work through algorithm design and use of caches.

3.3 Parallel Comparison Sorting

In recent years, using special purpose accelerators such as the Cell BE, and the GPUs have yielded tremendous performance gains across application areas. Prominent examples include semi-numerical algorithms such as sorting, graph algorithms, image processing, scientific computations, [116, 102] and also on several irregular algorithms like [139, 108]. However, accelerator based computing has relegated the role of CPUs. Typically in current models of computation the CPU transfers the input and the program to the accelerator device, and gets the result from the accelerator.

Sorting is a problem of fundamental importance in Computer Science with a rich history of algorithm design, analysis, and engineering. Several parallel algorithms and their corresponding efficient implementations targeted at modern architectures including the Cell BE [15],GPU based works including [110, 81, 32, 14], the Intel MIC [111], are being studied. However, all of the above works utilize only a homogeneous device.

Given the importance and relevance of hybrid computing, in this work we propose a hybrid algorithm for sorting on a CPU+GPU platform. We specifically consider comparison based sorting algorithms for reasons of wide applicability to settings such as variable length keys, and database records. We extend the algorithm presented in [81], which is a natural extension of the standard quick sort [56], to operate in a heterogeneous setting. The basic idea of sample sort [81] is to choose $k-1$ pivots, or *splitters*, from the input list. The input list is then split into k disjoint lists each containing roughly n/k elements. Each of the sub-lists can be sorted independently. Typically, a recursive approach is taken to reduce the size of the sublists further.

We redesign the above approach so as to work with CPU+GPU hybrid platforms. Our implementation offers advantages such as balanced work allocation amongst the CPU and the GPU, minimal idle time, and minimal inter-device communication. On a dataset of 64 M keys selected uniformly at random, our hybrid implementation offers a 40% speed-up over GPU based implementations of sample sort [81], and a 20% speed-up over the recent merge sort based work [32] when run on a CPU+GPU platform consisting of an Intel i7 980 and an Nvidia GTX 580 GPU. We also experiment with another hybrid platform consisting of an Intel Core2 E4700 with an Nvidia GT 520 GPU that resembles a commodity desktop configuration. On this platform, our algorithm shows an improvement of 18% compared to the currently best known comparison sort [32] results on GPU. We then extend our work to implement a variable key sorting algorithm which performs on an average 25% better than the current best known implementation proposed in [32]. Our work therefore shows that our approach has applicability to not only research-end devices but also commodity platforms which have a large user base. In Figure 3.1 and 3.2, we see the performance of our key-value sorting algorithm in the high-end and low-end platforms. The descriptions of the high-end and low-end platforms are provided in Chapter 3.

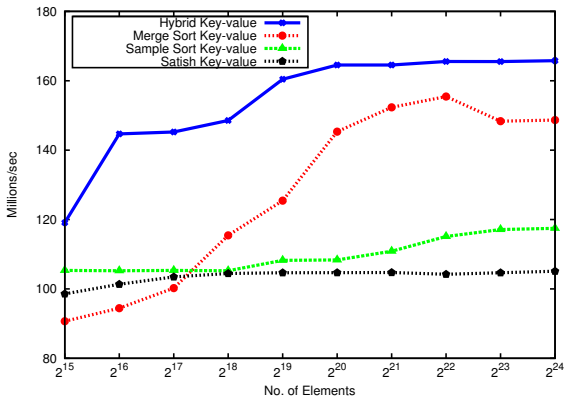

Figure 3.1 Performance on key-value pairs in high-end platform.

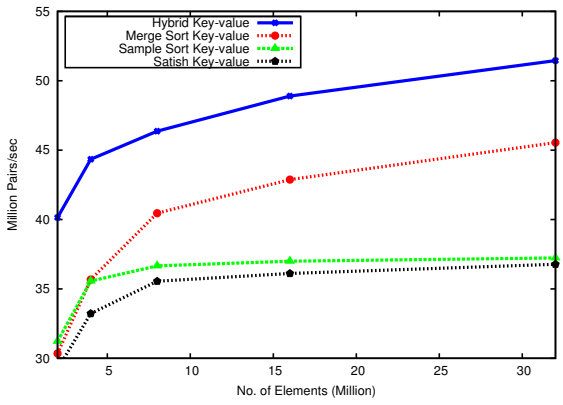

Figure 3.2 Performance on key-value pairs in low-end platform.

Table 3.1 Comparison of properties

PRNG	On-Demand Supply	Scalable	High Quality	Speed Rank
glibc `rand()`	×	√	×	5
CURAND	√	√	×	4
CUDPP	×	×	√	3
M.Twister	×	√	√	2
Hybrid PRNG	√	√	√	1

3.4 Hybrid Pseudo Random Number Generator

Randomness is an essential computing resource for many computations [90, 73, 66]. Hence, investigations into sources of high quality (pseudo) random number generators (PRNGs) are important. In parallel computing, designing parallel random generators is a challenging problem. This problem becomes more significant, as we are witnessing a shift to multicore processors.

Most of the pseudo random number generators based on GPUs however suffer from several drawbacks. For instance, PRNGs on GPUs require the application to pre-generate and store a large batch of random numbers and then use them in the application. Apart from occupying a significant portion of the limited storage on GPUs, this is not a satisfactory solution since the randomness demand of every application cannot be known apriori. In Table 3.1, we can see the properties of the current PRNGs that are commonly used and that of our hybrid PRNG.

It is therefore important that an on-demand pseudo random number generator be available so that each thread running on a GPU can make an API call, such as the `rand()` function in ANSI C [70], to obtain a new pseudo random number as required. Such an on demand generator also does not require as much storage to store the random numbers in the GPU memory. Secondly, another limitation of present generators on the GPUs is that they are not resource efficient. While the generator is working on the GPU, the host to which the GPU is attached, typically a multicore CPU, is computationally idle. This is not a good practice as the computational power of multicore CPUs is also ever increasing.

Our main result of this work is to design a high quality, fast, scalable, and on-demand random number generator. We achieve this by employing random walks on expander graphs. Each thread performing the walk is essentially executing independent of other threads. Therefore, our generator is thread-safe. We can see a plot of the performance of our generator in Figure 3.3.

To improve the performance of our generator, we employ a hybrid computing platform consisting of a multicore CPU and a GPU. Our generator produces 0.07 GNumbers per second. The results of our generator has been put through rigorous quality testing using test suites such as the DIEHARD battery of tests [83] and the TestU01 [78] suite.

We also show how to use our PRNG in two applications: list ranking, and a Monte Carlo based photon migration. These applications demonstrate the speed of generation and the quality of the hy-

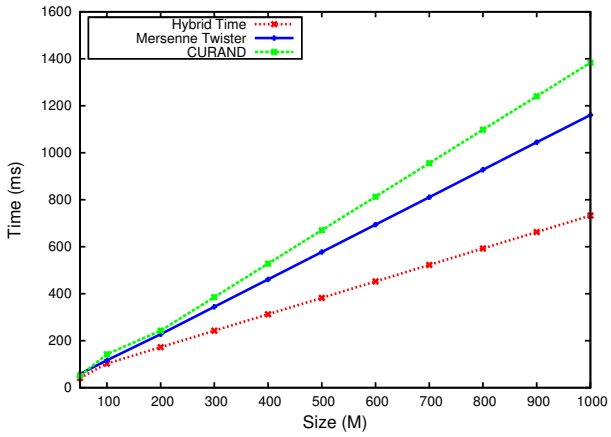

Figure 3.3 Performance over variation of sizes generated.

brid PRNG respectively. In both these applications, using our PRNG leads to reduced runtime, and improvements in quality.

3.5 Parallel List Ranking

List ranking a popular computing primitive. A very early work at parallel list ranking was proposed by James Wyllie in his Ph.D. thesis [141]. The biggest contribution of Wyllie's work was that of Parallel Random Access Machine (PRAM) model [40]. In that model the author proposed the use of a shared memory which can be accessed by several processors and can be used for sharing data and also synchronizing between themselves. In this thesis he applied the PRAM model for solving the list ranking problem using a technique called "pointer jumping". Pointer jumping is a very fundamental operation in parallel computation where any node can be made to point to the parent of its predecessor until convergence. This operation finds application in a wide variety of problems. In [141], Wyllie also discusses several other problems that can be solved using the PRAM model by adapting conventional sequential algorithms.

List ranking on GPUs was first proposed by Rehman et al. in [108]. In this work, the authors proposed a GPU optimized list ranking algorithm based on the popular Helman-Jaja list ranking approach that was proposed in [52]. The authors proposed a recursive Hellman-Jaja algorithm for the GPU which can rank 32 million elements in a second wand achieved a speedup of almost 9x over the parallel CPU implementation and around 4x over the best reported Cell Broadband Engine implementation that was proposed by Bader et al. in [11].

Wei and Jaja published a work on list ranking [139] that implemented list ranking on GTX 200 series GPUs. This result was in turn an improvement over and earlier work on list ranking [108]. Wei and Jaja employed a randomized algorithm to perform the list ranking where several elements were binned into a certain number of bins which were in turn allocated to each of the SMs of a GPU. This would

lead to each of the thread blocks completing the ranking process independent of each other and finally consolidating to provide the final results. The authors showed a performance of around 300 ms for a random list of 64 million elements.

This work was later improved by us [16] using an hybrid mechanism where we employed fractional independent sets to engage the CPU and the GPU both at the same time. The algorithm utilized asynchronous data transfer between the GPU global memory and the CPU memory. The iterative algorithm took a finite number of steps to reach $n/log(n)$ of nodes in the list. The rest of the nodes were removed with proper book-keeping. After this the smaller list was ranked on the GPU using a popular GPU algorithm for list ranking which took a very small fraction of the total time. After this step, the initially removed nodes were re-inserted into the list in the reverse order of their removal. Overall, the algorithm performed almost 25% better than the conventional GPU algorithm. This result also gives a clear indication that the hybrid approach is significantly better than the homogeneous accelerator based approach. In Figure 3.4, we see the evolution of our hybrid implementation over ther first accelerator based implementations that was proposed in [108] and [139]. We provide the details of the hybrid implementations in Chapeters 6 and 5 respectively.

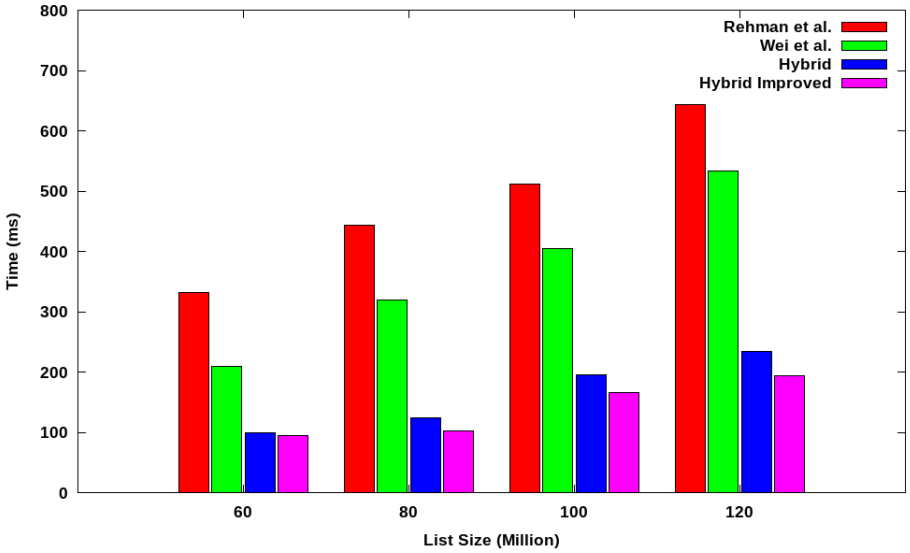

Figure 3.4 Improvement of List Ranking from 2009 to the current hybrid implementation. We can see that we acheived a speedup of mearly 3x over the original GPU implementation.

3.6 Graph Connected Components

Graphs are an important data structure in Computer Science because of their ability to model several problems. Some of the fundamental graph problems are graph traversals, graph connectivity, and a spanning tree of a given graph. In this work, we study the fundamental graph problem of connected components of a graph on the GPU.

Connected components is considered an irregular memory access algorithm (irregular algorithm), which is not a good for the GPU computational model which relies heavily on regularity of memory access. The focus of any algorithm designed for the GPU relies on regular/coalesced memory accesses and increasing computation, focusing on data movement in the shared memory. The requirements for connected components and GPU computational model are thus orthogonal to each other. Thus mapping the connected components algorithm on to the GPU is non trivial.

In [16], we also worked on a graph connected components problem. This is an extension of the work done in [123] where the authors have used the `O(nlog n)` Shiloach-Vishkin (SV) algorithm [119] to compute the connected components of a random graph. In this work, the authors employ several GPU optimizations in order to adapt the well known SV algorithm to the GPU. The optimizations like edge-hiding, pointer jumping and hooking of subtrees are critical contributions in the GPU computing and are now widely applied in other parallel implementations of graph problems. The authors of [123], demonstrated good speed-ups of around 9x over equivalent CPU implementations.

We now experiment on the work partitioning path of hybridization where we first partition the graph statically using a certain threshold and then perform the best known algorithm to compute connected components on each of the device. These algorithms execute in an overlapped fashion and then synchronize. This is then followed by a consolidation step where the results of both the devices are checked to provide the final result. Due to this non-blocking execution of the kernels in the two devices, it is pragmatic to say that the hybrid approach is always a better option where there is ample data parallelism. Our implementation on a CPU+GPU hybrid platform achieves an average speed-up of 25% compared to the best possible GPU implementation [123]. In Figure 3.5, we see the performance of our implementation over that of [123]. We also notice that our hybrid algorithm has very minimal idle time. We also show that our approach can lead to auto-tuning.

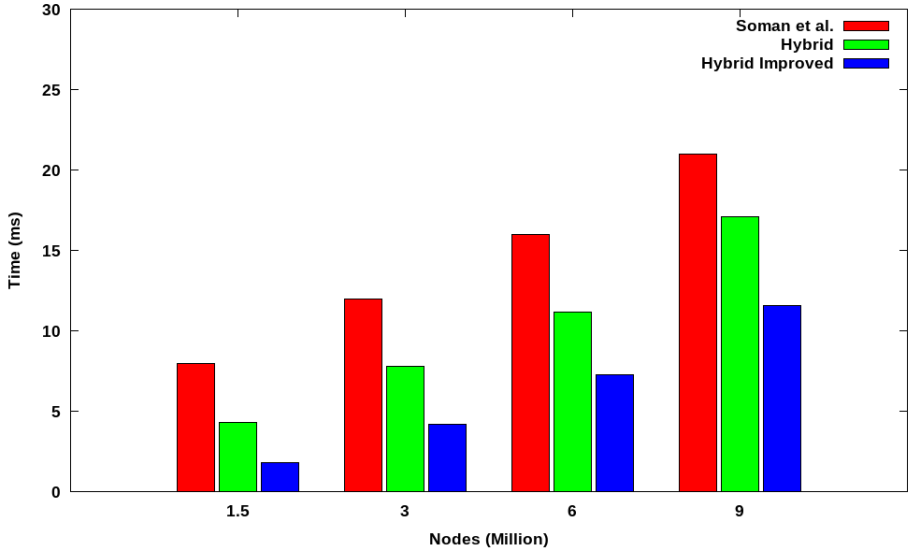

Figure 3.5 Time comparison of connected components with respect to the results shown by Soman et al. in [123].

3.7 Breadth First Search

In recent years, graph algorithm design has gained an important role in science, as many emerging large-scale scientific applications now require working with large graphs in distributed networks. Breadth-First Search (BFS) is of particular relevance because it is widely used as a basis for multiple fields. Other common graph applications also use Breadth-First Search as a fundamental part of their algorithm. Some of the relevant problems include flow network analysis, the shortest- path problem, and other graph traversals, such as the A* algorithm.

Disregarding memory optimization strategies, previous- graph parallelization efforts have been oriented toward masking the I/O problems with high doses of aggressive parallelism and multi-threading. Cray XMT, IBM Cell/BE, and NVIDIA GPUs are architectures that exploit such advantage and prioritize bandwidth over latency. Work on the mentioned platforms has shown great performance improvements in overcoming the high latencies incurred during graph explorations. The general purpose GPU (GPGPU) architectures have the added value of being an affordable solution while maintaining high throughput and low power consumption levels. While any of the previously mentioned platforms offers massive parallel processing power, its performance while traversing a graph will ultimately depend on its connectivity properties, the architecture, and the memory- subsystem. GPGPU architectures yield unmatched performance if sufficient parallelism is available and the graph fits on the GPUs memory. But they fail to yield the same performance otherwise, due to large overheads and the impossibility of overlapping the communication latencies with effective computation.

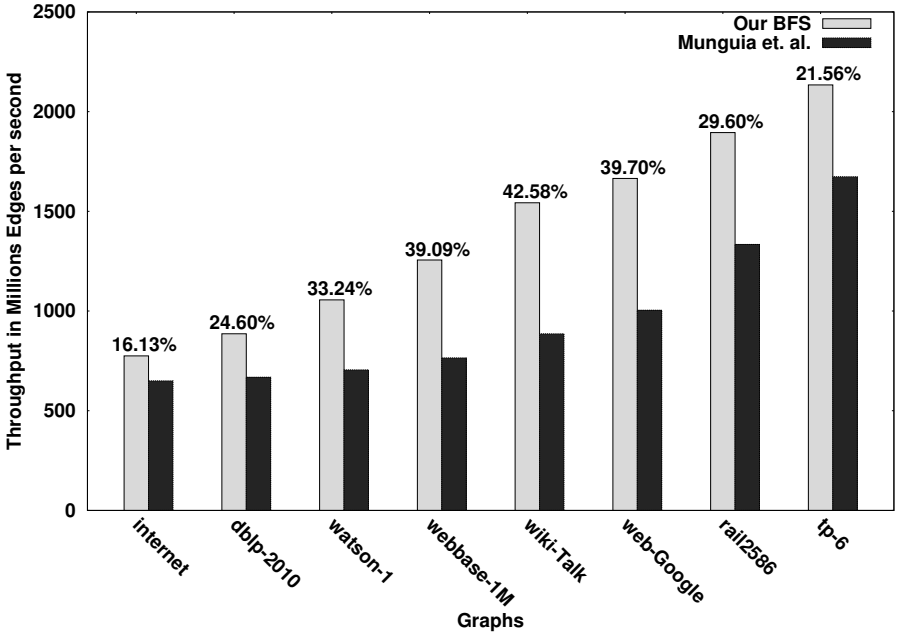

Figure 3.6 Performance of BFS on the UFL graphs obtained from [2]. The percentages indicate the improvement over the results of [91] over the same instance.

In this work, we envisage a new strategy for optimal graph explorations through graph pruning. In many recent works such as that of [37, 103], the authors have demonstrated the utility of graph pruning in high performance and parallel applications. We show that new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multicore architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. Graph pruning aims to reduce the size of the graph by pruning away certain elements of the graph. The required computation is then performed on the remaining graph. The result of this computation is then extended to the pruned elements, if necessary.

For performing BFS on modern generation graphs, we perform a graph pruning phase where we remove a majority of the pendent nodes that are present. We first show that such a preprocessing phase can help reduce the size of the graph by an average of 35% on a wide variety of real-world graphs. This helps us to obtain an average of 40% speed-up compared to the best known implementations for the above problems on similar platforms [91]. In Figure 3.6, we show the results of our implementation.

Our preprocessing simply involves removing pendant nodes from the graph. This is done iteratively so that nodes on pendant paths are also removed during preprocessing. In the post-processing phase, we show that extending the output of the computation on the smaller graph can be done in a very straightforward and quick manner.

Our results improve the state-of-the-art for graph BFS by 35%. We achieve an average throughput of 2 billion edges per second on a wide range of data sets including graphs from the University of Florida collection [2], and graphs generated using the Recursive Matrix Model (R-MAT). The R-MAT generator is efficiently implemented in the GTGraph suite[13].

PART I

Semi-Numerical Algorithms

Introduction

In recent years, using special purpose accelerators such as the Cell BE, and the GPUs have yielded tremendous performance gains across application areas. Prominent examples include semi-numerical algorithms such as sorting, graph algorithms, image processing, scientific computations, [116, 102] and also on several irregular algorithms like [139, 108]. However, accelerator based computing has relegated the role of CPUs. Typically in current models of computation the CPU transfers the input and the program to the accelerator device, and gets the result from the accelerator.

It is anticipated that advances in multi-core CPU technology will further drive the performance of CPUs so as to scale challenges such as the power wall, the memory wall, and the like. Further, it is strongly believed that future generation computer systems shall be heterogeneous in nature with a mix of multi-core CPUs and special purpose accelerators. Hence, it becomes important to design and implement efficient algorithms that work seamlessly on heterogeneous systems.

As CPUs evolve, there is a class of applications for which higher performance can be achieved by using both the CPUs and the GPUs in the computation. This intuition is further strengthened by the fact that the GPU is not a stand-alone device and actually needs a CPU to host it. Further, special purpose accelerators are not ubiquitous in nature and hence pushing such accelerators to be so may not be the right choice. Hence, it is required that one make a judicious choice of tasks that one wishes to solve on a given kind of architecture. It is also believed that in future, high performance computing will be dominated by platforms that contain a heterogeneous mix of processors with varying capabilities.

The challenge then lies in making the right choices so as to make best use of heterogeneity to achieve fast, scalable, and efficient solutions to problems of practical interest. We explore these challenges in this Part of the thesis where we explain our solutions for Comparison Sorting and Pseudo Randomness with applications. In Chapter 4, we talk about hybrid solution of comparison sorting. In Chapter 5, we discuss Pseudo Random Number generation on parallel architectues and use an application to show its usability.

Chapter 4

Hybrid Comparison Sorting

4.1 Introduction

Sorting is a problem of fundamental importance in Computer Science with a rich history of algorithm design, analysis, and engineering. Several parallel algorithms and their corresponding efficient implementations targeted at modern architectures including the Cell BE [15],GPU based works including [110, 81, 32, 14], the Intel MIC [111], are being studied. However, all of the above works utilize only a homogeneous device.

Given the importance and relevance of hybrid computing, in this work we propose a hybrid algorithm for sorting on a CPU+ GPU platform. We specifically consider comparison based sorting algorithms for reasons of wide applicability to settings such as variable length keys, and database records. We extend the algorithm presented in [81], which is a natural extension of the standard quick sort [56], to operate in a heterogeneous setting. The basic idea of sample sort [81] is to choose $k-1$ pivots, or *splitters*, from the input list. The input list is then split into k disjoint lists each containing roughly n/k elements. Each of the sub-lists can be sorted independently. Typically, a recursive approach is taken to reduce the size of the sublists further.

We redesign the above approach so as to work with CPU+GPU hybrid platforms. Our implementation offers advantages such as balanced work allocation amongst the CPU and the GPU, minimal idle time, and minimal inter-device communication. On a dataset of 64 M keys selected uniformly at random, our hybrid implementation offers a 40% speed-up over GPU based implementations of sample sort [81], and a 20% speed-up over the recent merge sort based work [32] when run on a CPU+GPU platform consisting of an Intel i7 980 and an Nvidia GTX 580 GPU. We also experiment with another hybrid platform consisting of an Intel Core2 E4700 with an Nvidia GT 520 GPU that resembles a commodity desktop configuration. On this platform, our algorithm shows an improvement of 18% compared to the currently best known comparison sort [32] results on GPU. We then extend our work to implement a variable key sorting algorithm which performs on an average 25% better than the current best known implementation proposed in [32]. Our work therefore shows that our approach has applicability to not only research-end devices but also commodity platforms which have a large user base.

4.1.1 Motivation

We observe that in most GPGPU based computing, the CPU is practically idle in the computation process. This leads to inefficient resource usage, more so as the computational power of present generation multicore CPUs is on the rise. Hence, to improve performance, we use such a hybrid CPU and GPU system and target full resource utilization. We call this as *hybrid multicore computing*, or *hybrid computing* in short. Hybrid computing is gaining tremendous research attention of late given that issues such as power and performance dominate parallel computing.

Further, in a recent influential work [79], the authors argue and provide evidence for showing that on a diverse collection of 14 workloads, GPUs can offer only modest performance advantage compared to multicore CPUs. Sorting is one of the workloads considered in [79] where it is shown that the GPU is on an average only 1.5 times faster than the CPU. We interpret the message of [79] as not to compare one device against the other, but to study the benefits of using both the devices simultaneously. We call this as *hybrid computing*.

Intuitively adding more computational resources should lead to faster execution. However, as the resources in our model are heterogeneous in nature, apart from algorithm design, one has to address challenges such as load balancing, appropriate work distribution, and synchronization issues so as to translate the gains into reality. One useful technique in this context is that of static work partitioning that is used in several popular libraries such as ScaLAPACK [18]. This technique requires one to decompose the computation into independent subtasks so that overheads associated with communication and synchronization can be minimized.

In line with the above, to be able to make effective use of a hybrid platform with respect to the sorting workload, one has to think of algorithms that can quickly create independent subproblems. A portion of these independent subproblems can then be solved on the CPU and the remaining subproblems can be solved on the GPU. One such sorting algorithm that can create independent subproblems is quick sort [56]. A generalization of quick sort where we choose several pivots, called as *sample sort*, has been shown to be highly efficient on the GPU [81]. We therefore consider extending sample sort as a hybrid algorithm. While the results of [81] are recently improved by a merge sort based work in [32], merge sort does not give rise to independent subproblems. Further, the merge step requires several data transfer and synchronization steps when run in a hybrid setting. Hence, we note that such algorithms are not efficient on hybrid platforms. Our hybrid sample sort actually outperforms the results of [32].

While it is noted in several works that GPUs are very amenable to highly data parallel operations. Radix sort is one such computation that benefits significantly from GPUs [110, 14]. Efficient implementations of radix sort on GPUs such as Thrust radix [110] and [14] can offer a throughput of 300 Million elements per second (Meps) for 32 bit inputs. This is not matched by comparison based sorting on GPUs, the fastest of which has a throughput of 200 Meps. However, radix sort on the other hand suffers from some specific drawbacks. In the first place, it is highly dependent on the efficiency of each of the operations such as insertion and deletion of the digits. Also, they are not easily adaptable to different types of data such as variable length keys or multi-field records. Additionally, radix sort suffers

from a worse space complexity when compared to comparison sorting techniques. These are some of the factors that motivate us to work more towards comparison sorting algorithms rather than the radix based ones.

4.1.2 Related Work

Radix sort algorithms are some of the most efficient algorithms that have been implemented on GPUs and other multicore architectures. Radix sort is one of the easiest algorithms to be implemented in parallel machines because of its reducibility to a popular primitive which is the *scan* or parallel prefix operation. In works such as [19, 145, 125], the authors have shown the use of the scan and split operation for efficiently implementing the radix sort routine. A popular randomized parallel algorithm for radix sorting was proposed by Helman et al. in [51]. The most popular recent implementation of radix sort was proposed by Merrill and Grimshaw [89]. However, radix sort algorithms suffer from a basic bottleneck, where the sorting process becomes computationally more expensive with the increase in the size of the keys.

In this Chapter, we present a comparison sort algorithm, which is the other well known mechanism for sorting fixed length as well as variable length keys. As comparison sort depends on a comparison function defined by the user and can have an ample parallelism involved, there has been extensive work in this area since the earliest days of GPU computing. A result using bitonic sort on GPU platform was proposed by Peters et al. in [104]. The popular divide and conquer based quicksort [56] algorithm also has been of high interest in parallel computing. It was first demonstrated on GPU by Sengupta et.al. in [116]. The result of this work was further improved in [23]. Another prominent result in comparison sort was proposed by Satish et. al. [110]. In this paper, the authors have shown a merge sort algorithm for the GPUs where small chunks of split data are put on the GPU and are then merge using popular merging techniques efficiently.

Merge sort is another popular sorting algorithm which recursively merges multiple sorted subsequences into a single sorted sequence. The first parallel merge sorting algorithm was proposed by Richard Cole in [26]. In the earliest works on merge sorting on the GPU, the work of Purcell et al. in [107], is of high importance. The current best result in comparison sort on GPUs is by Davidson et al. [32]. The work also provides several insights into efficient implementation on GPUs by reducing memory access latencies, improving register utilization and reducing segmentation.

Sample sort is a randomized version of the GPU quick sort technique where, instead of dividing the entire list into two parts at any given iteration, there are a set of splitters that are chosen which divides the list into many parts. Each of the sublists can be then processed in parallel irrespective of the other bins. Some of the earliest works proposed in parallel sorting are those in [20, 36, 61]. In these works sample sorting has been implemented using randomly chosen s samples from the n/p elements that are present in each processor. Then every sth elements has been chosen as the splitter. Each processor P_i then performs binary search on these splitters for each of the input elements which is then used to bin the values. After this step, local sorting is done in each of the bins. Hence, it is natural that the biggest

bottlenecks in this implementations are that of gathering the samples and sorting them. The larger the value of s, the better the load is balanced, but it also significantly raises the gathering overhead. Another bottleneck is that, no matter how the binning process is done, it creates higher irregularity and in turn effects the efficiency. In the solution provided by Helman et al. in [51], the authors choose n/p^2 splitters using a mechanism that incurs no extra overheads in the identification of the splitters. They employ a oversampling strategy due to which a higher number of bins are created and hence increases the locality which reduces irregularity to a higher extent. In a more recent work, sample sort routine using random splitters was shown by Leischner et.al in [81]. Sample sorting was also implemented by Dehne et al. in [41] using deterministic splitters rather than random ones.

All of the above divide-and-conquer techniques gives rise to several efficient many-core algorithms which vary only in the way they are individually implemented. In the gcc sort routine MCSTL proposed in [121], the authors show that each core of the processors get an equal portion of the input. Each of these portions are then sorted using an algorithm called the Introsort proposed in [92]. Then a final k-way merging step is done for to merge all the intermediate steps.

There have been some efforts towards the design of hybrid algorithms for sorting. Notable among them is the work by Govindaraju et al. [45], where the authors use a radix-bitonic combination algorithm for sorting. Also worth mentioning is the work of Sintorn and Assarsson [122], where the authors show a quicksort-merge sort technique for sorting a massive list of real numbers. These works can be classified as hybrid only because of the mixed algorithms that the authors use and not because of the simultaneous execution on two heterogeneous processors.

4.2 Our Solution

In this section, we describe our hybrid sorting algorithm. As mentioned in Section 4.1.1, we use a hybrid variant of sample sort [81]. While merge sort may be more amenable on pure GPU solutions, as mentioned in [110], we prefer sample sort as it avoids the costly inter-device communication. The main idea of sample sort is to divide the input list into independent sublists using $k = O(\sqrt{n})$ pivots (also called as splitters). These k splitters are chosen uniformly at random from the input list. The sublists can be sorted recursively until their size drops below a threshold. Each such small sublist can then be sorted individually. A summary of the algorithm appears in Algorithm 1. The algorithm is described pictorially in Figure 4.1. Figure 4.2 shows an example run of the algorithm.

In the following, we now describe the changes required for executing the algorithm in a hybrid manner. One important factor that our algorithm design addresses is to aim for load balance between the CPU and the GPU and also within the GPU. In all our algorithms, we have used labels such as `CPU`, `GPU` and `CPU→ GPU`. The label `CPU` refers to computations that take place in the CPU, and `GPU` refers to those on the GPU. The label `CPU→ GPU` refers to the data transfers that is happening between the CPU and the GPU.

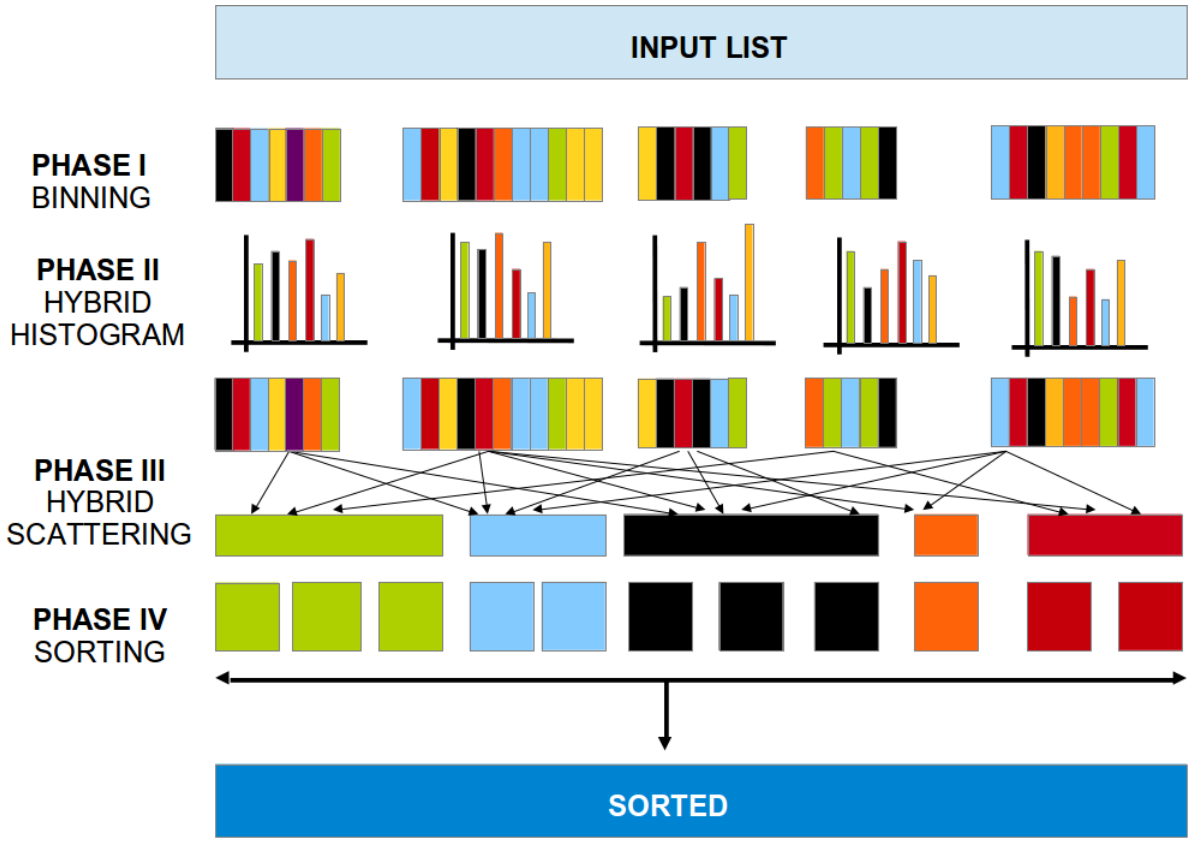

Figure 4.1 Different phases in hybrid sorting. The different colors represent the different bin labels which are brought together by scattering.

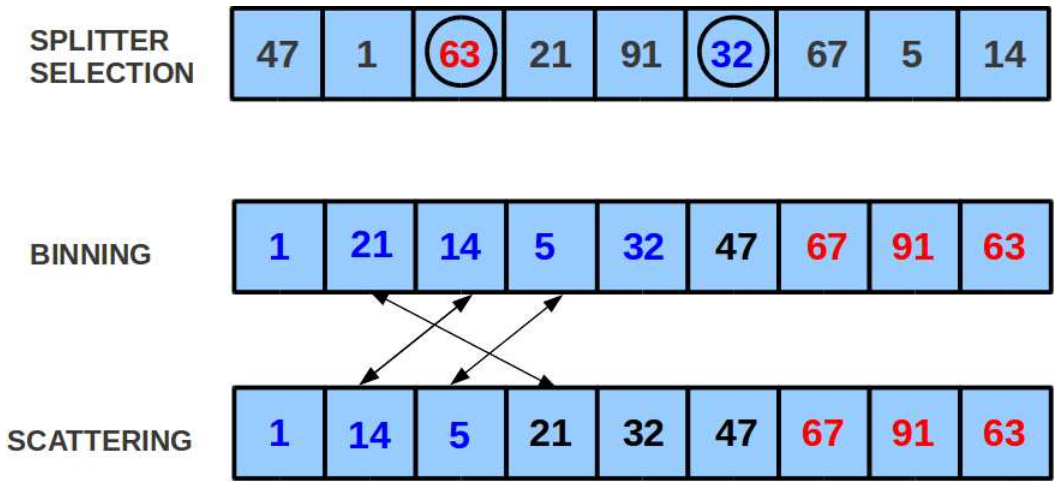

Figure 4.2 An example run of our algorithm on a sample input of 9 elements. In this, we show the first pass of the algorithm on our input list, that creates the first set of bins.

Algorithm 1 HybridSort()

1: ***PHASE I :*** Creating bins.
2: ***PHASE II :*** Hybrid Histograms on CPU and GPU.
3: ***PHASE III :*** Scattering elements on both CPU and GPU.
4: ***PHASE IV :*** Recurse and sort small bins.

4.2.1 Phase I

Phase I involves selecting splitters as a uniform sample of the input list and then identifying the bin into which each element can be assigned to based on the splitters. As in [81], we build a binary search tree of the splitters. The bin to which an element belongs to can then be efficiently identified by searching in the binary search tree. Having such a binary search tree is efficient since it reduces thread divergence of a block of threads in a GPU. Such techniques have been found to be useful in also multicore CPUs [81]. As we see in Algorithm 2, we now partition the list of labels into two parts each of which can be handled by the CPU and the GPU respectively. Let us call these lists as L_c and L_g respectively. As of now we employ the static partitioning strategy.

The static scheme of distribution of work have often been used in several popular benchmarks such as ScaLAPACK [18]. The static strategy has been found to be good because of its optimal communication costs, lesser synchronization overheads and scalable load balancing properties. In Section 4.4, we show how the threshold of separation of the work between the two devices have a bearing on the final result. Also, this parameter helps in another way by enabling us to auto tune the entire application.

After the separation of the data between the two devices, each of them now completes the binning process by using the binary tree formed using the splitters. To facilitate a hybrid execution of the other phases, we now associate each element with its bin number. We call these bin numbers as *labels*. This allows us to treat some of the following phases on inputs in the range $[1, k]$ thereby simplifying the later phases.

Algorithm 2 Phase I($Integer\ *I, Integer\ n, BlockSize\ BLOCK$)

1: ***PHASE I : Create Bins***
2: Select elements from the list at intervals of $\sqrt{n}$ and from a binary search tree. n is the input number of elements.
3: `CPU` **::** Set Threshold which determines the ratio of division of L
4: `CPU` **::** Divide L as L_g for GPU and L_c for CPU
5: `CPU`$\rightarrow$ `GPU` **::** Transfer L_g to GPU
6: `CPU` **::** Use tree to optimally divide L_c among several bins in overlap of transfer.
7: `GPU` **::** Use tree to optimally divide L_g among several bins after transfer is complete.

4.2.2 Phase II

Algorithm 3 Phase II($Integer * L_c, Integer * L_g, BlockSizeBLOCK$)

1: ***PHASE II : Hybrid Histograms***
2: **For** each list L_g and L_c do in parallel
3: `CPU` **::** Compute histogram H_c of L_c for $LEN/BLOCK$ elements
4: `GPU` **::** Compute block-wise histogram H_g of L_g for $LEN/BLOCK$ elements
5: **endfor**

At the end of Phase I, elements that have a common label are scattered across the input. In Phase II, described in Algorithm 3 for each label, we count the number of elements that have this label. This is done by computing the histogram of the list L_c on the CPU and the histogram of the list L_g on the GPU. In our application we use histograms in the second phase for getting the frequency of each bin label that is present on each block.

4.2.3 Phase III

Algorithm 4 Phase III($Integer * H_g, Integer * H_c, BlockSizeBLOCK$)

1: ***PHASE III : Scattering***
2: **For** each histogram L_g and L_c do in parallel
3: `GPU` **::** Perform local scan on H_g to get local offsets
4: `CPU` **::** Perform global scan on H_c to get global offsets
5: **endfor**
6: **For** each List L_g and L_c do in parallel
7: `GPU` **::** Perform local scattering to all bins in each $BLOCK$
8: `CPU` **::** Perform global scattering across the single block
9: **endfor**

Phase III rearranges the elements of the input list according to their labels. This essentially is a scattering step wherein elements with a given label are grouped together from various initial locations. As we see in Algorithm 4, our approach in this phase involves performing a prefix sum across the histogram results of all the blocks. The prefix sum gives us the intra-block offsets for each of the labels. We first perform an intra-block scattering so that all the bin labels are adjusted to their appropriate places within the block. Following this, we then scatter the labels across blocks. The CPU exploits the non-blocking nature of the GPU kernels by executing a similar operation on its own part of the data. As, the CPU works with only a single block of threads, the scattering carried out is a global scattering rather than a local block wise scattering as done by the GPU. The difference between the local and global scattering being that of scattering across multiple blocks and a single block of threads. As all of the L_c labels are present on the CPU on only a single block, the global scattering can happen in an overlap with the GPU scattering.

4.2.4 Phase IV

Algorithm 5 Phase IV($Integer * L_g, Integer * L_c, BlockSizeBLOCK$)

1: ***PHASE IV : Sorting individual bins***
2: `CPU→ GPU` **: :** Transfer CPU bins to GPU
3: Repeat PHASE I through III until each BIN small enough for local sort.
4: Separate the bins again among CPU and GPU and apply best known sorting technique on each bin.

In Phase IV, described in Algorithm 5 we sub divide each of the bins by recursively repeating Phases I to III as long as the size of each bin is not small enough to be efficiently sorted by a single thread of the GPU. This arises from the fundamental bottleneck that the GPU suffers from where, we cannot assign a heavy compute intensive job to each of the threads of the GPU. Hence, after the first round of splitting and scattering, the size of each bin is still of the order of $\sqrt{n}$. This creates a high compute load for each of the threads of the GPU which needs a smaller sized bin. So, in this phase, each of the bins are again subjected to the same splits from Phase I. After, the size of each bin reaches a suitable size, they can be efficiently sorted by a single thread. This smaller bin size is discussed in the following section.

4.3 Implementation Details

We borrow the some of the implementations for the initial 3 phases from [81] to improve the efficiency of the code on the GPU side. However, the hybrid nature of our algorithm introduces other implementation challenges and opportunities for optimizations. This section describes those in each of the phases.

4.3.1 Phase I

The computation in this phase involves finding the bin number to which each input element belongs to. The bin number of an element is the number of splitters that are smaller than the element. On the GPU, if all the elements are stored in contiguous locations, then one can benefit from a large number of coalesced accesses in this computation. A further optimization described in [81] builds a height balanced binary search tree out of the splitters. At this point, finding the bin number of an element translates to a search in the binary search tree of splitters. One can also reduce thread divergence using this technique [81].

For the above reasons, we notice that indeed this phase can be executed entirely on the GPU. In fact, the time taken when this phase is executed on the GPU entirely is less than 2% of the overall time even for large inputs. However, we choose to perform this step also on the CPU and the GPU. This is justified by the fact that the input array is available only at the CPU at the beginning. For the GPU to start executing, the input array has to be made available at the GPU. In addition, since the other phases run simultaneously on both the CPU and the GPU, the output of this phase has to be sent to the CPU.

This involves an unnecessary data transfer step, which can be avoided if the bin numbers of a portion of the input is computed on the CPU.

We therefore choose a certain threshold and split the input array I into two parts, I_c and I_g. We choose splitters in I, and also find the bin numbers of elements in I_c on the CPU. The array I_g and the splitters are simultaneously transferred to the GPU. The GPU then computes the bin numbers of elements in I_g. At the end of the Phase 1 we have a list of bin labels that correspond to each of the input elements and is now called L_g and L_c for the GPU and the CPU parts respectively.

4.3.2 Phase II

Phase II computes the histogram of the bin labels. Computing histograms on the GPU is a well-researched problem. Briefly, the entire computation is split into computing local histograms at each of the SMs and then combining these histograms to arrive at a global histogram. One of the factors that affect the GPU performance is the number of threads that should be launched on the GPU so as to use to entire bandwidth on the GPU. Within each SM, to compute a local histogram, threads use shared memory to improve memory coalescing. A fundamental bottleneck in the histogram computation on GPUs is that GPUs offer very low throughput when using atomic operations.

This however, helps the hybrid computing model, since as described in Algorithm 3, the CPU also computes the histogram on an independent input, L_c. This computation is data is simply added using atomic primitives supported by OpenMP [24]. As we are having a constant block size, it is comparatively simpler to parallelize across all the available cores of the CPU. A manual unrolling of the histogram serves this purpose and uses all the six cores that are available on the CPU and also gives us a significant performance benefit. In addition to this, we ensure to carefully optimize the histogram computation of the histogram on the CPU. We read in a certain tile of data into the L2 cache of the CPU. This tile size is based on the size of the L2 cache on the Westmere CPU. We iterate in steps of this tilesize so that the entire data on the L2 cache is used and does not require to be used afterwards. Inside each of the unrolled loops, we now use a second tile size that reads data from the L2 cache into the local cache of each of the cores. These cores now maintain a histogram store on the shared local cache and performs the atomic increments. Each of the instructions issued for increments are vectorized so as to ensure proper SIMD execution. We ultimately get a bandwidth bound performance from the CPU which is marginally better than the GPU. At the optimal threshold, the GPU stays idle for only 2% of the entire Phase II time.

At the end of this histogram computation, we synchronize the two devices using the CUDA event synchronization functions. This is required since Phase III can start only after the histogram computation is completed on both the devices.

4.3.3 Phase III

The computation in this step is a scatter operation. On the GPU, we compute the offsets of elements using a prefix scan over the results of the histograms. One thing to watch out here is that the scattering step involves lots of irregular memory accesses on the GPU. In order to reduce this irregularity, and facilitate higher coalescing, we use a special storage format for the histograms from Phase II. Instead of conventionally storing the labels of the histograms in a row-major format as is commonly used, we use a column major format that allows all the similar bin labels to be stored contiguously. The GPU performs the local scattering across each of the blocks after which, we get all the bin labels arranged in contiguous groups. The original input elements are also scattered along with their bin labels this so that the list of input elements can be partitioned into independent sublists. We discuss in detail about this optimization strategy in Section 4.3.5. We note that a similar strategy is used in [102].

The CPU scattering process also proceeds in parallel to the GPU scattering. As, we know that irregular operations like gather and scatter adds a high overhead to the performance of each device, the split of work between the two devices for this step provides a high benefit. The CPU scattering also proceeds in a similar way as that of the GPU, with the only difference being that, the CPU works with only a single block of threads and hence is technically doing a global scatter rather than a local one. We optimize the scattering process on the CPU by again performing a two level tiling on the CPU block so that maximum use of the shared cache can be ensured.

4.3.4 Phase IV

At the end of Phase III, the bins are independent of each other and can be sorted independently. In Phase IV, each of the bins are recursively subdivided until they are small enough to be sorted quickly by a single GPU thread. The threshold is chosen based on typically the number of elements that could be accommodated into the shared memory of the GPU. On the Hybrid-High platform (cf. Section 2.9.1), the GPU has a shared memory of 64 KB. In our implementation, each thread block on the GPU has 1024 threads. Given this, the threshold is chosen as 64 elements. Similar calculations can be done for the GPU on the Hybrid-Low platform from Section 2.9.2. The thread wise quicksort that is performed by the GPU is done in the shared memory. Typically for a list of 16 million, three rounds of recursion was required after which the sorting on GPU took less than 2 ms to complete. At the above threshold, the histogram and scattering phases in all of the recursive computations consume 5% of the total time (See also Figure 4.4).

So far, we have discussed implementation issues surrounding our algorithm on the CPU and the GPU. Apart from the above mentioned implementation issues, hybrid algorithms have other issues to address that arise due to the hybrid nature of the computation. This could be in the form of additional data structures, additional synchronization overheads, and the like. These may introduce critical scalability challenges and hence minimizing their effect on the workload is needed. In the following section, we mention some of these challenges with respect to the hybrid sorting algorithm.

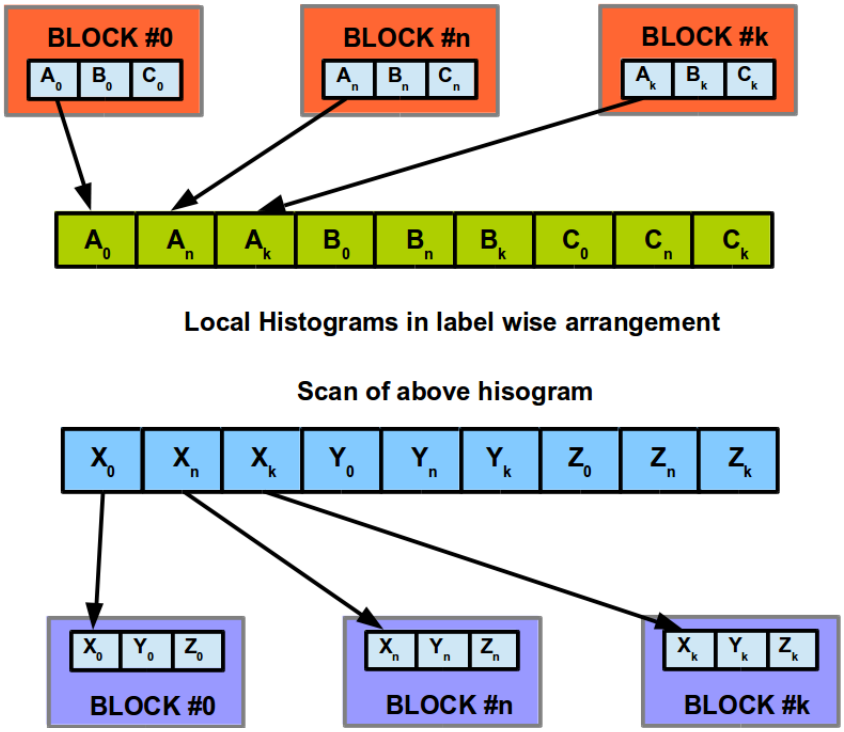

Figure 4.3 The count for each bin label is computed through histograms in shared memory and is written back in column major order to global memory. Then the prefix sum provides the offsets for each of the labels.

4.3.5 Memory Usage

A vital resource in hybrid computation is the available memory on the GPU for the hybridization. Often there is a need to use extra data structures for the purpose of book keeping, synchronization and consolidation. This extra space that is used may have an adverse effect on the scalability of the code and is hence important to reduce or reuse as much as storage as possible. In this implementation, there are two vital phases of hybrid computation. One is that of the histogram computation and the other is that of the scattering.

In the histogram phase, the GPU is computing its local histograms which is stored in a single $d\_histStore$ space that is $\mathcal{O}(BLOCK * BIN_G)$. Here BIN_G is the total size of the bins that are allocated to the GPU. The CPU on the other hand operates on only a single block and hence is $\mathcal{O}(BIN_C)$, where BIN_C is the bins formed out of the CPU part of the elements. Each of these data structures are hence consuming a large chunk of the available memory, and it is needed to reduce the footprint that is left by these data structures. Once this phase of the computation is completed, in Phase III, the scattering step needs an additional data structure, that will store the scattered elements from $d\_blockHistStore$. That will again require a space of the same order of the histogram store. In order to optimize the usage of the available memory on the GPU, we do the following.

4.3.5.1 Higher coalescing of reads

In Figure 4.3 we see that the histogram is conventionally stored in the form of an array where each block sequential writes the entries for $A_0, A_1, A_2...A_n$. The numbers in this case represents the bin labels which varies from A_0 to BIN_G. The drawback of this is that when the scattering phase starts, it becomes an impediment to read all the A_0s from each of the block entries. As, the A_0s are all stored at $BLOCK$ sized distances from each other, there is a big stride that the thread needs to make which introduces a lot of cache misses in the process. In order to overcome this, each of the clockwise histograms are computed in the shared memory of the GPU and is written back on to an array called $d\_blockHistStore$ in the GPU global memory. This helps keep all the A_is together in the first block, all the B_is in the second block and so on. This facilitates a higher coalesced read during the scattering phase.

4.3.5.2 Reuse of histogram store

Now, after all the 0 bin labels have been read from the $d\_blockHistStore$ structure, the first $BLOCK$ size of the $d\_blockHistStore$ get freed. It is now available for reuse during the scattering phase. So, what we now do is that during the scattering, we simply read all the bin labels from each of the blocks, and scatter to the same locations from which the labels were just read from. This helps in both a coalesced read and a write. Additionally, there is no need for an extra structure to store the scattered elements. This step alone helped us in getting a big performance benefit over the conventional histogram storage.

4.3.6 Sorting variable length keys

While sorting strings there are a completely different set of challenges that we come across. In this case, we first start from the implementation of the sorting of the key-value pairs. We select the first 32 bits of the string as the key and the rest of it as the value. In the global memory we store each of these keys and pointers to the rest of the strings. In the case of a tie, we need to go out into the global memory and keep fetching sequences of 32 bits until the tie is broken.

In the hybrid model, there is a huge heterogeneity that is involved in the two devices that are involved. In the CPU, while sorting strings, the overhead for fetching secondary bits for breaking of the ties is a relatively easier phase because of the low latency of fetch from the main memory. On the other hand the latency of fetch from the global memory is comparatively much higher in the case of the GPU. It must also be noted that when a tie occurs there is a certain amount of time for which the other threads have to wait until the tie is resolved. Now, in a massively parallel architecture like the GPU, this accounts for a high overhead. Also, the idle time of the GPU increases which effectively affects the efficiency of the entire process. So, in order to resolve this, we adopt a certain optimization, where we keep an additional data structure maintaining 32 additional bits of the strings on the GPU memory at all times. So, in the final sorting step, in case we face a tie, the additional bits can be immediately looked up and the tie

can be resolved without the other threads not waiting for a large amount of time. In this way, what we effectively achieve is a higher efficiency of the GPU which now stays idle for a lower time period. Also, because we are storing only pointers to the rest part of the strings, this additional data structure only makes the sorting equivalent to a 64 bit key-value sort, which does not affect the scalability to a large extent.

4.4 Experiments and Results

In this section, we describe our experiments and results. We use the experimental platforms described in Chapter 3 for all our experiments. We use the label "Hybrid-High" for the hybrid platform introduced in Section 2.9.1 and the label "Hybrid-Low" for the hybrid platform introduced in Section 2.9.2. In our experiments, we run our implementation and the implementation from [81, 32] on the GPUs included in our hybrid platforms. The software for the implementations of [81, 32] have been provided by the authors of the respective papers. For experimenting with our implementation, we have broadly concentrated on two different types of data sets. One is the fixed length keys which are of general interest to anyone designing a new sorting algorithm. The other is the sorting of variable length keys which is built over the implementation of the fixed length keys. The sorting of variable length keys such as strings is not that much focused on by most of the implementations, but is of very high importance and finds a very wide variety of applications. These applications can be in databases, data analytics, intelligent systems and such. We primarily experiment using the following different types of inputs.

1. **Fixed length keys:** For fixed length keys, we use several different types of real numbered data sets for bot 32 bits and 64 bits. Each of the data sets are inspired from the experiments that have been carried out in [51].

    - **Uniformly Random:** This dataset corresponds to that of an input drawn uniformly at random and where each element is 64 bits long. For generating the dataset, we use the GPU Mersenne Twister algorithm [87]. We seed the algorithm from [87] using a random value generated using the `glibc rand()` and passed to the routine.
    - **Gaussian Distribution:** The Gaussian/normal distribution the numbers are distributed on a space or real numbers with a certain mean and standard deviation. Using the Central Limit Theorem, we can sum up a certain number of random numbers and take their average to get numbers that follow the normal distribution. In order to do that we make 12 calls to the glibc `rand()` and take their average.
    - **Key Value Pairs:** Another important study that is required in any sorting application is the results on key-value pairs. We associated 32 bit random values to each of 32 bit the keys that we are sorting and recorded the performance.
    - **Deterministic Duplicates:** In this experiment we tried the use of deterministic duplicates which is a distribution of natural numbers with a block wise logarithmic progression. Let

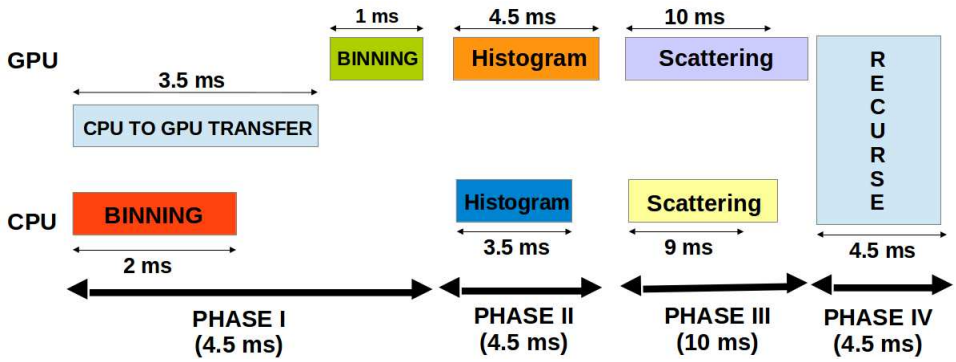

Figure 4.4 Phase wise timing diagram for 4 million elements.

the input of n numbers be broken into p blocks for p being a power of 2. The first $p/2$ blocks have all the numbers as $\log n$, the next $p/4$ blocks have all the numbers as $\log(n/2)$, and so on. In general, all the numbers in the block $p/2^i$ are set to $\log(n/2^{i1})$, for $i = 1, 2, \cdots, p-1$. Finally, for the pth block, the first $n/2p$ elements are set to be $\log(n/p)$, the next $n/4p$ elements are set to $\log(n/2p)$, and so on.

- **Randomized Duplicates:** This input is a slight modification of the deterministic duplicates dataset. Let the input of n numbers be arranged as p blocks. Each block is filled as follows. Let B be an array of size r with each element of B chosen uniformly at random from $[0, r-1]$. Let S be the sum of elements in B. Then, the first $B[1]/S \times n/p$ elements of the block are chosen uniformly at random from $[0, r-1]$, the next $B[2]/S \times n/p$ elements of the block are chosen to be another number uniformly at random from $[0, r-1]$, and so on. In our work, we set r to the total number of cores available on the CPU and GPU combined.
- **Staggered:** In this the input of n numbers is arranged in p buckets. In the buckets numbered 1 to $p/2$ (both inclusive), all the numbers are chosen uniformly at random from $[(2i-1) \cdot 2^{31}/p, (2i) \cdot 2^{31}/(p-1)]$, where i is the bucket number. For buckets numbered $p/2+1$ to p, all the elements are chosen uniformly at random from $[(2i-p-2) \cdot 2^{31}, (2i-p-1) \cdot 2^{31}/(p-1)]$.
- **Bucket Sorted:** In this an input of n numbers is organized into p buckets as follows. The first n/p^2 elements in each bucket are chosen uniformly at random from $[0, 2^{31}/p - 1]$, the second n/p^2 elements in each bucket are chosen uniformly at random from $[2^{31}/p, 2^{32}/(p-1)]$, and so on.

2. **Variable length keys:** For sorting strings we mainly use two different data sets. First we experiment with a Protein Sequence Database [132] which is 650 MB in size and has many protein sequences that are represented using the popular "FASTA". In this format, each of the sequences are up to 120 characters long. Apart from this, we also experiment with a data set of 500 MB size that is containing a set of random strings and are up to 500 characters in length.

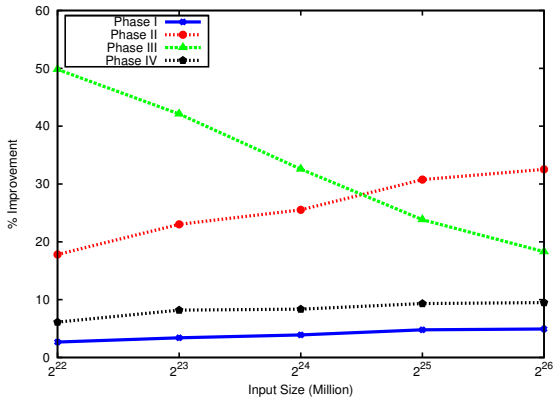

Figure 4.5 Percentage improvement over sample sort [81] at various phases.

4.4.1 Profiling, Resource Utilization, and Idle Times

One aspect of our hybrid implementation is that each phase of our hybrid algorithm runs faster than the corresponding phase in the pure GPU version from [81]. This is shown in Figure 4.5. In Figure 4.5, the times for Phase II and Phase III are only over the first iteration. This is justified since the overall time taken by Phase IV, which includes recursive calls to Phases I-III, is a very small portion of the entire runtime of the implementation. Also, this recursive calls happens over data that is present in each of the bins that has been created in the first iteration. Hence, the overall impact of the hybridization is felt mostly in the first iteration of the application. In Figure 4.5, we show the percentage improvement of our hybrid implementation over the pure GPU sample sort implementation. We now analyze the results of Figure 4.5.

In Phases II and III, we notice that there is a significant improvement that is achieved over the corresponding pure GPU phases. In both of these phases, we notice that there is a benefit of around 25%. Further, the improvement of Phase II increases as the size of the input increases. This is because of two reasons. Firstly, as the input size increases, the distribution of the numbers happen over a larger number of bins. Hence, the conflicts arising during the atomic computations are reduced. Also, the larger number of bins are now also divided among the CPU and GPU which leads to a higher degree of work distribution. This leads to a better works sharing and consequent increase in the gain of Phase II.

In Phase III, we notice that the percentage improvement decreases over the input size. This can be attributed to the nature of the computation in Phase III. The scattering operation introduces a lot of irregularity in the memory accesses on a GPU. Such uncoalesced access patterns are known to affect GPU performance. However, the performance gain can be attributed to the fact that CPUs do not suffer as significantly as GPUs on gather-scatter operations.

Notice from Figure 4.5 that performance gain of our hybrid implementation in Phase I and Phase IV is not very significant. This can be explained as follows. Phase I is not highly compute intensive. Indeed in our experimentation, we notice that Phase I if run entirely on the GPU takes only less than 1% of the total time. However, even this phase is done in a hybrid manner as that reduces the data communication

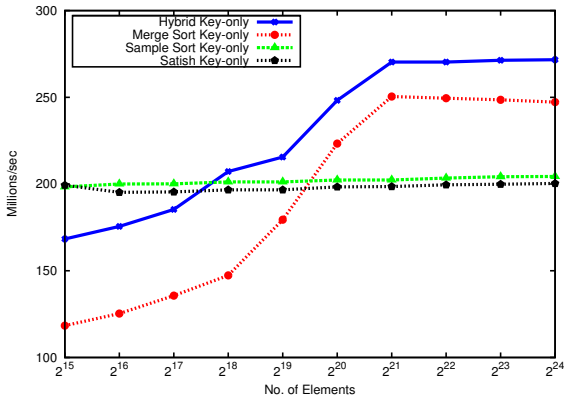

Figure 4.6 Performance on uniformly random keys sorting in high-end platform.

overhead in the later phases. Doing so has the other benefit that the CPU utilization improves, and hence the idle time reduces. This can be noticed also from Figure 4.4 where the phase-wise timings are shown for an input of size 4 M. We define idle time of a device as the total time for which the device is idle. The idle time of an implementation is then the maximum idle time experienced by any of the devices. In our algorithm, the GPU is idle for at most 5% of the total runtime, and the CPU is idle for at most 12% of the total runtime. So, the idle time of our implementation is 12%.

In Phase IV, the size of each of the bins reduces significantly which again leads to a reduction on the compute that is required by either the CPU or the GPU threads.

4.4.2 Results of Sorting

4.4.2.1 Results on Fixed Length Keys

In this section, we show the results of our implementation on various experimental datasets of fixed length keys. Since most of the GPU sorting algorithms are designed on 32 bit numbers, we have experimented using the 32 bit integers as well. Results on 64 bit integers are shown later.

The input is generated as a uniformly random dataset described earlier. In Figure 4.6, we report the performance of the 32 bit **key only sorting**. In this case, we see that we achieve almost a 23% improvement on an average over the closest best known result [32].

We now look at the sorting results for **key-value pairs** in Figure 4.7. As can be noticed from Figure 4.7, our hybrid implementation is on average 40% faster than the result of [81], and on average 20% faster than the result of [32]. This improvement can be attributed to the fact that the majority of the computation involves operations such as histogram and scattering which are very amenable to a hybrid execution environment.

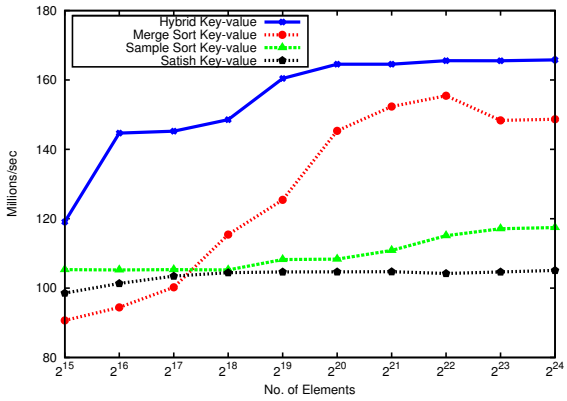

Figure 4.7 Performance on key-value pairs in high-end platform.

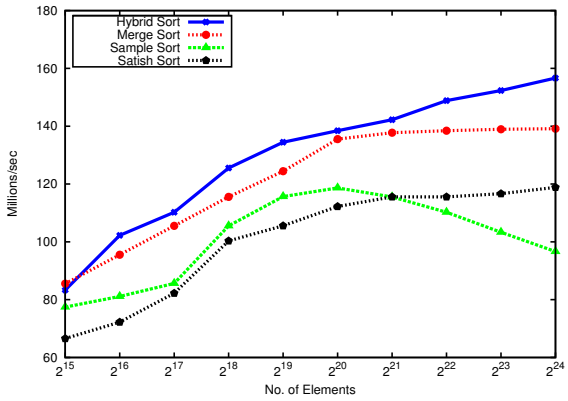

Figure 4.8 Performance on 32 bit Gaussian input.

We report the behavior of our algorithm on the inputs such as Gaussian distributed, Deterministic Duplicates, Staggered, and Bucket sorted in the following. In Figure 4.8, we see the performance of our algorithm on the Gaussian distributed input. We achieve an improvement of 10% on an average on the Gaussian input because of the added overhead in the scattering phase. As the number of bins created in this case is higher than the other inputs, the scattering in the first phase of the algorithm consumes more time. In both Figure 4.7 and Figure 4.8, we notice a significant performance improvement only when the input size cross the threshold of 2^{18} elements. This can be attributed to the fact that on inputs greater than 2^{18} elements, the number of sublists that are created before the threshold of sorting is reached enables the GPU to achieve bandwidth saturation. Hence, the number of threads employed towards the whole GPU scattering process are higher which helps get higher gains.

In Figure 4.9, we see the performance of our algorithm on the input of Deterministic Duplicates. In this dataset, the inputs are in logarithmic progression across all the blocks of the GPU. This leads to a more idealistic data set for our mechanism and hence provides a good performance. We see that we achieve almost a 15% speedup over merge sort in this case at the best case.

In Figure 4.10, we see the performance of our hybrid sorting mechanism, on the staggered input, where we observe that our sorting method suffers a decline in performance. This decline can be attributed to the fact that, the staggered input is randomized block wise over a certain period. Due to this, there is a higher degree of irregularity that is introduced. This leads to a higher overhead in the CPU-GPU scattering phase and hence suffers a overhead. In the other sorting algorithms such as merge sort, this scattering phase is not there and hence performs significantly better.

The performance on Bucket sorted input is reported in Figure 4.11. In this case too we see that the performance declines because of the block wise randomness. This case is similar to that of the staggered input and hence suffers from a significant overhead in the scattering phase. The performance of other

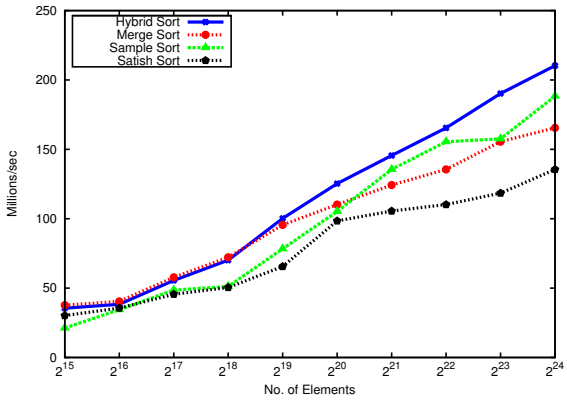

Figure 4.9 Performance on 32 bit Deterministic Duplicates input.

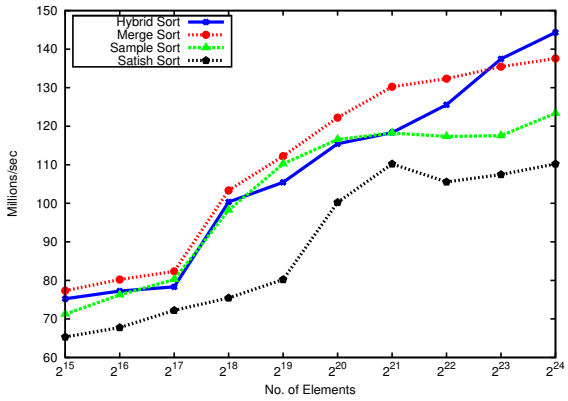

Figure 4.10 Performance on 32 bit staggered input.

algorithms where such kind of irregularity does not come into the picture, performs good. In Figure 4.12, we see the performance on the input of Randomized duplicates where the performance is around 17% better on an average than the other algorithms. In this case too, the presence of duplicates, the irregularity is decreased and hence the consequent overheads are also reduced to a good extent.

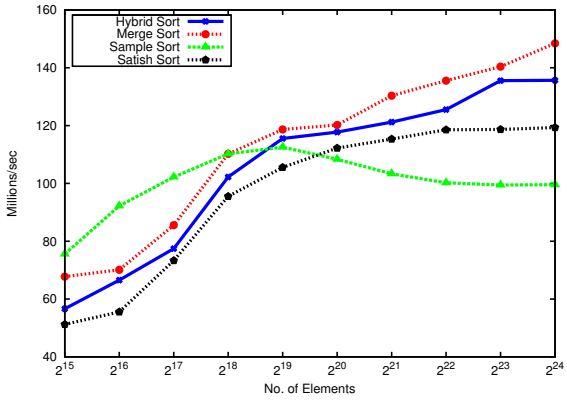

Figure 4.11 Performance on 32 bit bucket sorted input.

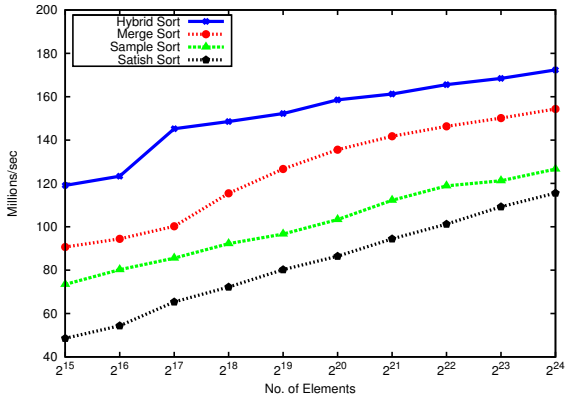

Figure 4.12 Performance on 32 bit randomized duplicates input.

4.4.2.1.1 64 Bit Inputs Performance of our sorting algorithm on 64 bit inputs is also important to be experimented with. With the increase in bit length of the input, the stability of radix based sorting algorithms decreases and hence the real advantage of comparison based sorting mechanisms become important. The impact has been already studied at length in the work by Leischner et.al. in [81].

In Figure 4.13 and 4.14, we see our performance in the 64 bit sorting case in comparison with the sample sort. We achieve a benefit of 18% on the Gaussian input and of around 20% on the case of the Deterministic Duplicates.

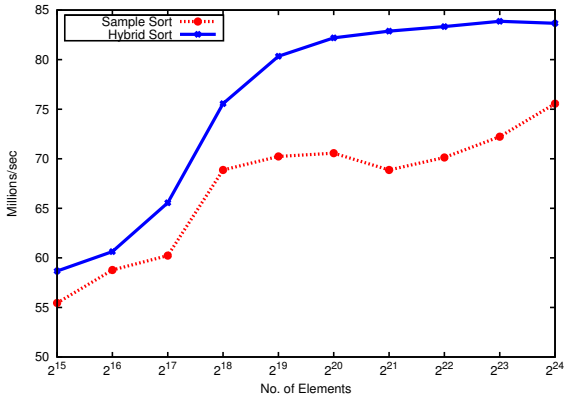

Figure 4.13 Performance on 64 bit gaussian input.

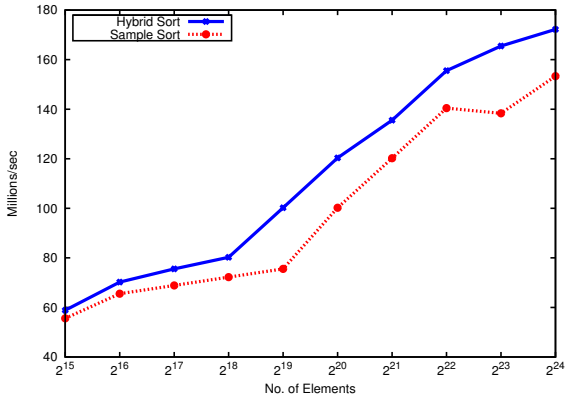

Figure 4.14 Performance on 64 bit deterministic duplicates input.

4.4.2.1.2 Results on Other Platforms We also experimented on the Hybrid-Low platform described in Section 2.9.2. We see the result of the sorting in Figure 4.15. We experiment using 64 bit key-value pairs on this platform and achieve a benefit of 18% on an average over the best known merge-sort implementation.

In the Figures 4.16, 4.17, we see the performance of our key only and key-value sorting across three different platforms that we have already explained in Section 2.9. The sorting method performs the best with the K20c GPU as the GPU has a bigger sized L2 cache and offers a much higher degree of

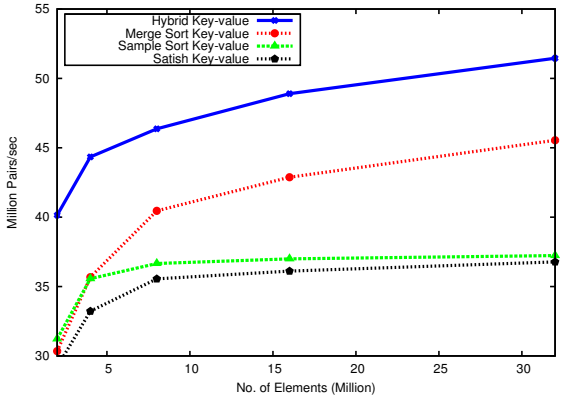

Figure 4.15 Performance on key value pairs on low-end platform.

parallelism. So, due to the L2 caching of the data, the overhead suffered during Phase III is significantly offset.

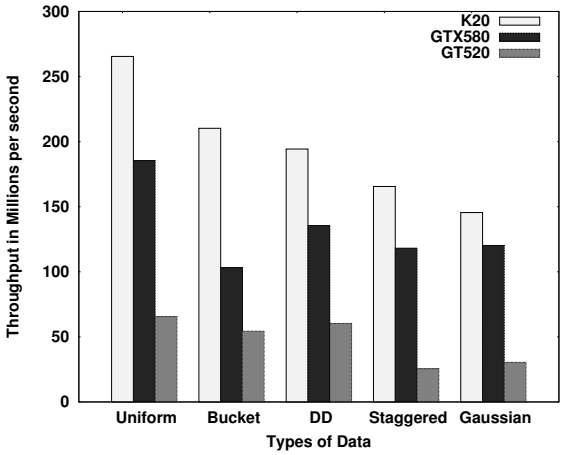

Figure 4.16 Performance of 32 bit sorting across different platforms at input of 2^{21} uniformly random elements.

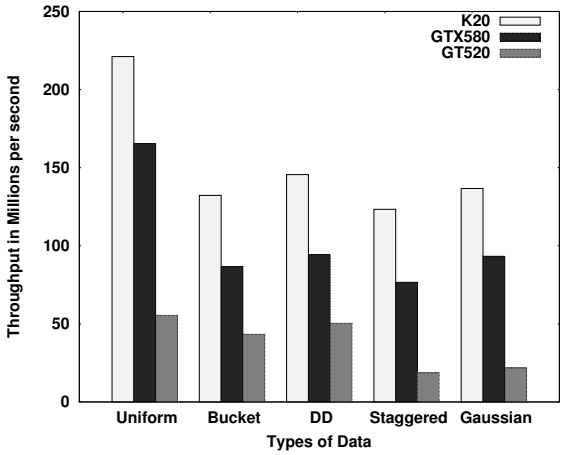

Figure 4.17 Performance of key value sorting across different platforms at input of 2^{21} elements.

4.4.2.2 Variable Length Key Sorting

We now focus on the experimentation of our variable length key sorting. The most natural data sets that we can obtain for variable length keys are that of strings. We experimented on broadly two categories of strings, one of which is synthetic and the other one natural. The synthetic data set was generated using randomness that is offered by popular PRNGs like the Mersenne Twister. We report the performance in Figure 4.18. This is the result that is obtained from our execution on random string database. We get a benefit of about 24% over the current best known sorting result that was reported by Davidson et al. in [32]. One of the reasons for achieving the performance is the cached reads for the additional 32 bits that are kept for resolving the ties that might occur in the sentences. In the protein database that we choose, there is a good possibility that there can be repeats of protein sequences for considerable distances. In Figure 4.19, we can see the performance of our sorting algorithm with the caching and without the caching. It is important to mitigate the cost of divergence that will be caused because of this ties. The cached reads helps in lowering the cost of divergence by not eliminating it, but rather distributing it amongst all the threads that are at work. We can see that because of the caching of the look forward bits, we get almost 2x benefit. As the size of the caches are limited, we can store only up to 32 bits for each SIMD warp to work on.

Another possibility that we tried out, is storing random 32 bit sequences from the entire sentence. However, choosing 32 bit random sequences demand another kernel which uses a standard linear con-

gruential generator for randomness. This puts an additional overhead that affects the overall performance.

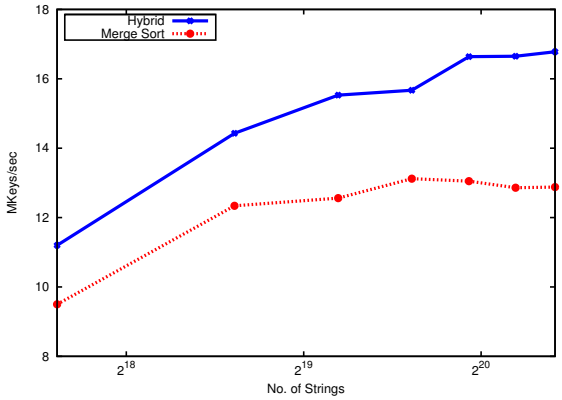

Figure 4.18 Performance on variable length random strings.

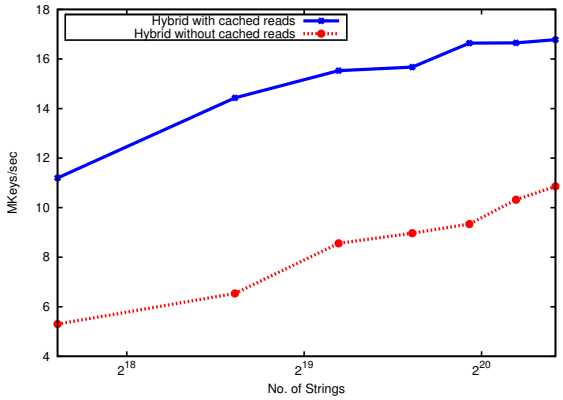

Figure 4.19 Performance on protein database strings with caching.

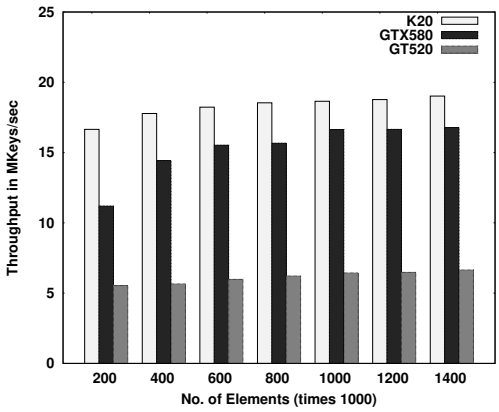

Figure 4.20 Performance of variable length key sorting across different platforms.

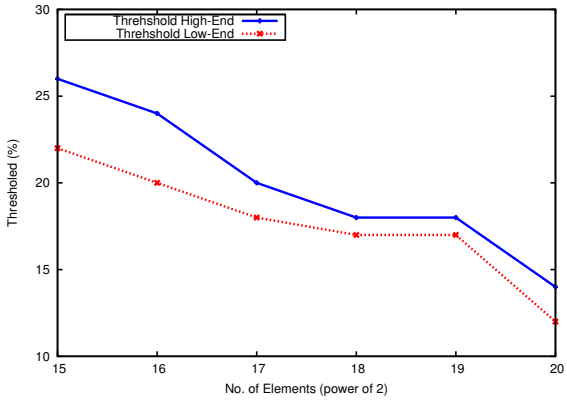

Figure 4.21 Variation of threshold.

In Figure 4.20, we see the performance of the string sorting on random strings on the several platforms of experimentation. As this sorting routine is an extension of the key-value sorting, we can see that the performance on the K20c is the best where we can ideally use the caches for storing the look-ahead bits. The performances on the other two platforms are also efficient and the GPU and CPU are both busy in compute for nearly 90% of the entire run time. However, as we can notice from Figure 4.20, that there is no significant increase in the performance on the low end GT520 platform. This can

be attributed to the lower number of threads that the GPU is capable of running. The amount of gain that can be extracted from the platform is obtained at the lowest problem size of 200,000 elements. Bandwidth saturation is achieved via the number of subproblems that are created at that input size. With the increase of input size, even though we obtain similar speed ups, there is no major change as the GPU is already at its peak performance limits.

4.4.3 Threshold Variation

Recall that in our implementation we use a fixed, static threshold for dividing the work between the CPU and the GPU. Such static thresholding has been shown to be beneficial as it reduces the communication overhead apart from lesser synchronization overheads and scalable load balancing properties. Further, even though the computation in each phase of our algorithm can potentially have a different threshold, such a phase-wise thresholding again introduces communication overheads. Therefore, we use a common threshold across all phases. The threshold values are experimentally determined. We first create a static partition of the values with 10% of the bins on the CPU and the rest on GPU. Post this step, we vary the threshold over a sufficiently big range so as the determine the optimaal split between the two devices.

In Figure 4.21, we see the variation of the threshold percentage that we chose for the static partitioning strategy. We notice that the threshold of partitioning steadily decreases with the increase in the input size of the elements. This can be explained as follows. As the size of the input increases, a higher threshold means that the CPU compute time does not evenly match the GPU compute time. Hence, a lower threshold is suitable. A similar kind of a behavior is observed in the case of the low-end platform too. The only difference being, the lower range of variation in the threshold because of a higher ratio in GPU to CPU peak performances.

4.4.4 Scalability

Our sorting approach demonstrates a good degree of scalability as can be seen from Figure 4.6. We were able to run experiments upto 2^{24} elements. However, as with the case of most experiments invlovlving GPUs, the limitation was the fixed size of the global memory. However, there were several modifications that were introduced as explained in Section 4.3.5. As a result of these modifications we were able to achieve 40% higher amount of scalability due to a higher efficient use of the available global memory space and the data structures that were used for the gather, scatter and histogram computations.

4.4.5 Results of Phase II

The computation in Phase II is essentially that of a histogram and hence a scalable and efficient hybrid histogram is of independent interest. Therefore, we compare the performance of Phase II of our

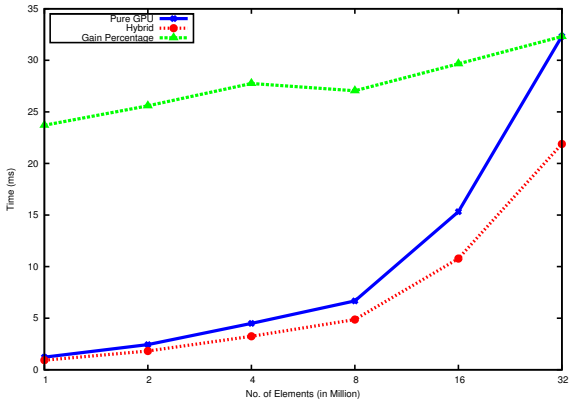

Figure 4.22 Time comparison with respect to pure GPU implementations and the performance gain.

hybrid algorithm with that of [106]. The work of [106] computes a histogram of the data on a GPU alone.

As is mentioned in several earlier works [79, 113], the performance of the histogram on the GPU depends heavily on the utilization of the shared cache of the GPU and the minimization of atomic conflicts. Further, the GPU is not very amenable to such atomic operations. Hence, this computation has the potential to benefit from a hybrid computing model. This is verified in Figure 4.22, where we see that there is a gain of nearly 25% on an average over the pure GPU implementation. Such a gain is remarkable given that the peak FLOP rating of the CPU in our experimental platform is only a tenth of that of the GPU.

4.5 Conclusion

Our hybrid sorting clearly demonstrates the benefits that can be gained out of the use of heterogeneous processors that are most commonly available in today's commodity desktops and laptops. In this work we have implemented and verified our algorithm agains a wide array of inputs of fixed length as well as variable length keys. All the results definitively show the adavantage of heterogeneous implementations. In the near future we will be having completely heterogeneous processors such as the ones of AMD APUs and Intel Ivybridge. Hence, in hybrid programming and research in this area holds a lot of promise.

In the near future we want to work on greater heterogeneous platforms such as the ones involving multiple GPUs and other multicore processors such as the Intel MIC. It will be interesting to see the performance of our sorting mechanism on such kind of platforms where there will be both tightly coupled and loosely coupled processors. It will be our goal to arrive at a efficient performance model for such kind of platforms using various computing primitves like sorting, searching, ranking and graph algorithms.

Chapter 5

Pseudo Random Number Generator for Hybrid Platforms

5.1 Introduction

Randomness is an essential computing resource for many computations [90, 73, 66]. Hence, investigations into sources of high quality (pseudo) random number generators (PRNGs) are important. In parallel computing, designing parallel random generators is a challenging problem. This problem becomes more significant, as we are witnessing a shift to multicore processors.

Most of the pseudo random number generators based on GPUs however suffer from several drawbacks. For instance, PRNGs on GPUs require the application to pre-generate and store a large batch of random numbers and then use them in the application. Apart from occupying a significant portion of the limited storage on GPUs, this is not a satisfactory solution since the randomness demand of every application cannot be known apriori. It is therefore important that an on-demand pseudo random number generator be available so that each thread running on a GPU can make an API call, such as the `rand()` function in ANSI C [70], to obtain a new pseudo random number as required. Such an on demand generator also does not require as much storage to store the random numbers in the GPU memory. Secondly, another limitation of present generators on the GPUs is that they are not resource efficient. While the generator is working on the GPU, the host to which the GPU is attached, typically a multicore CPU, is computationally idle. This is not a good practice as the computational power of multicore CPUs is also ever increasing.

In this chapter, we address these limitations and demonstrate the design and implementation of a fast, efficient, on demand, and high quality PRNG on a platform consisting of CPUs and GPUs.

5.1.1 Motivation

Most of the current random number generators are iterative in nature. The most popular among these are the linear congruential generators which was first proposed by Lehmer [80], matrix congruential generators [77], and the matrix recursive generators [72]. Several of these principles have been implemented to provide some widely used PRNGs in GPUs [87, 99].

Many of the popular PRNGs on the GPU, such as the Mersenne Twister [87] and CUDPP Rand [131], require an initial knowledge of the quantity of random numbers required. This poses a severe difficulty to applications where no such previous knowledge is available. Also, there may be applications which require random numbers on the fly during execution. The current PRNGs suffer from this disadvantage as the requirement cannot be changed dynamically.

Also, linear PRNGs, for example the `rand()` [70] function, were originally designed for sequential machines. Inherently it runs an linear congruential generator (LCG) [72] to generate new random numbers and requires seeding to produce different sequences on each run. So, each of the iterations are dependent on the earlier states for producing new numbers. Due to these state dependencies, they become unusable for highly multi-threaded architectures and are not thread-safe. We cannot call this generator from each core of a multicore processor as the generator would not produce correct results in such an execution.

From the above discussion, it is clear that existing PRNGs on GPUs suffer from several drawbacks. To motivate our work further, we present four properties that a parallel pseudo random number generator has to satisfy.

- **Scalability:** Scalability suggests that large quantities of random numbers can be generated without any limitations.
- **Quality:** Many cryptographic and security applications solely depend on good sources of random numbers [144, 93]. Hence, the quality of the generator is important.
- **On demand generation:** It must be possible to use the generator without a-priori knowing the quantity of random numbers required. Ideally, a simple API call should produce a new random number without a large overhead.
- **Performance:** The performance aspect suggests that the time spent on generating a random number is as small as possible. One common way to measure this is to study the number of random numbers that can be produced in a unit time.

We now give a comparison of the qualities that are possessed by the currently available PRNGs in Table 5.1. The speed ranking in Table 5.1 (rank of 1 is fastest), shows the relative time taken by each PRNG to generate a fixed quantity of random numbers.

We now turn our attention to the aspect of resource efficiency of PRNGs on GPUs. We observe that in most GPGPU based computing, the CPU is practically idle in the computation process. This leads to inefficient resource usage, more so as the computational power of present generation multicore CPUs is on the rise. Hence, to improve performance, we use such a hybrid CPU and GPU system and target full resource utilization. Hybrid multicore computing is gaining tremendous research attention of late given that issues such as power and performance dominate parallel computing. Arriving at an efficient hybrid PRNG that meets the requirements listed in Table 5.1 is a step in that direction.

Table 5.1 Comparison of properties

PRNG	On-Demand Supply	Scalable	High Quality	Speed Rank
glibc `rand()`	×	√	×	5
CURAND	√	√	×	4
CUDPP	×	×	√	3
M.Twister	×	√	√	2
Hybrid PRNG	√	√	√	1

5.2 Related Work

One of the early implementation of PRNGs for GPUs is the Mersenne Twister (MT) first proposed in [87] and later extended in [86]. Both these implementations however require that the quantity of random numbers to be pre-specified.

In [131], authors have given a source of randomness necessary for graphics applications based on the MD5 algorithm proposed by Rivest [28]. However, one major drawback of this is that CUDPP Rand usually do not scale to very large requirements.

5.2.1 Our Methodology and Results

Our main result of this work is to design a high quality, fast, scalable, and on-demand random number generator. We achieve this by employing random walks on expander graphs. Each thread performing the walk is essentially executing independent of other threads. Therefore, our generator is thread-safe.

To improve the performance of our generator, we employ a hybrid computing platform consisting of a multicore CPU and a GPU. Our generator produces 0.07 GNumbers per second. The results of our generator has been put through rigorous quality testing using test suites such as the DIEHARD battery of tests [83] and the TestU01 [78] suite. Our generator passes most of these tests as reported in Section 5.4.2.

We also show how to use our PRNG in doing a simple simulation of photon migration. These application demonstrate the speed of generation and the quality of the hybrid PRNG respectively. In both these applications, using our PRNG leads to reduced runtime, and improvements in quality.

5.3 Our Random Number Generation Technique

The main idea behind the development of our generator is to use parallel random walks on an expander graph. As each of the random walks on the graphs is entirely independent of each other, any thread of the GPU can make a request for random number(s) at any point of time. The only operation involved is to select a neighbor from the expander graph uniformly at random, perform a walk, and

return the destination node as a random number. This random selection of a neighbor can be made by using a few random bits. In the following, we explain our approach in more detail. Implementation details are presented in Section 5.3.2.

5.3.1 Expander Graphs

We now define an expander graph and also describe the expander graph we use in our construction. Let $G(V, E)$ be an undirected regular graph of degree d. For a subset of vertices $U \subseteq V$, let us denote by $(U, \overline{U})$, with $\overline{U} = V \setminus U$, the set of edges that have exactly one endpoint in U and another endpoint in $\overline{U}$. The edge expansion of G, denoted $\alpha(G)$, is defined as:

$$\alpha(G) = \min_{\substack{U \subseteq V; \\ |U| \leq |V|/2}} \frac{|(U, \overline{U})|}{|U|}.$$

A family of graphs $\mathcal{G} = \{G_1, G_2, \cdots\}$ is called an edge expander family if there exists a constant c so that for every $G \in \mathcal{G}$, $\alpha(G) \geq c$. So here, c represents the edge-expansion factor of the graph. If $|V| = n$, then ideally it would be possible to hae a c which is independent of n. In the ideal case, such kind of an independence will exist for every construction of an expander graph. Although in a majority of applications it is observed that for a constant c and every k, there exists a construction in the range of $k, k+1, ..., ck$. Also, it is preferrable that the degree d_n for every vertex slowly grows to n and is bounded by some constant. In most of the applications explicit constructions of expander graphs are extremely simple. It is possible to obtain an efficient construction from the strict upper bound and an adjacency list that can be obtained using a closed form formula.

Example : If we define a prime number p, and a graph $G_p = (V_p, E_p)$, in which $V_p = 0, ..., p-1$ and for $a \in V_p - 0$, then vertex a is connected to $a+1 \mod p$, to $a-1 \mod p$ and to its multiplicative inverse $a^{-1} \mod p$. So vertex 0 is connected to 1, to $p-1$ and has a self-loop. So, if we count all the self loops, we will see that the graph is 3-regular. As all the vertices 0,1 and $p-1$ has self-loops, we can see that the graph is a union of a cycle over $|V_p|$ and a matching over $p-3$ vertices $V_p - 0, 1, p-1$. So, there exists a constant h, for which each p in G_p has an edge-expansion of at least h. A sample execution for the same is shown for $p = 13$ in Figure 5.1.

In our construction of a PRNG, we have made use of explicit definitions of an expander graph due to Gabber-Galil [42]. Here we consider the graph G as a bipartite graph with two independent set of nodes X and Y. For a given integer m, with $n = 2m^2$, we can assign unique labels of the form $(a, b) \in \mathbb{Z}_m \times \mathbb{Z}_m$ to each of the vertices in X and Y. A Gabber-Galil expander on n vertices is defined as follows [42]. The vertices of the graph are tuples of the from (x, y) for $x, y \in \mathbb{Z}_m$. The neighbors of a vertex (x, y) in X can be found in Y by these labels : $(x, y), (x, 2x+y), (x, 2x+y+1), (x, 2x+y+2), (x+2y, y), (x+2y+1, y)$, and $(x+2y+2, y)$. All the above calculations are done modulo m. The expansion of the graph is shown to be $\alpha(G) = (2 - \sqrt{3})/2$ [42].

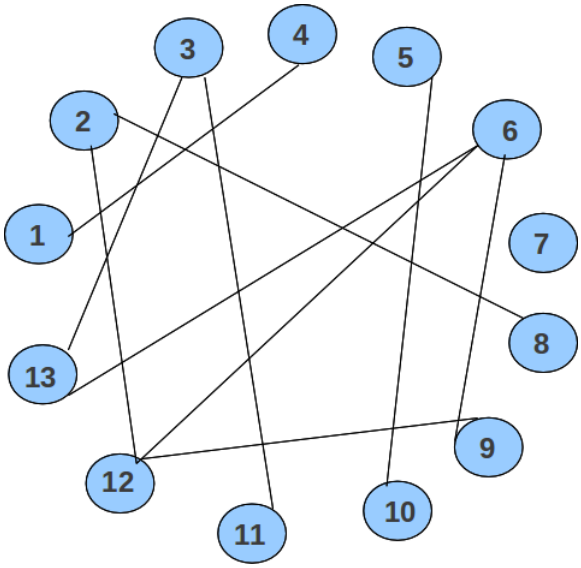

Figure 5.1 The expander graph G_{13}.

It has been shown [59] that random walks on expander graphs have a rapid mixing property so that the position of the walk after a certain steps is close to the stationary distribution of the underlying Markov process. For all our purposes, in this chapter, we do not statically create and store an expander. We use the functions given in the previous paragraph to create the vertices on the fly during the execution of the PRNG application.

5.3.2 Implementation Details

We initialize a 7-regular Gabber-Galil expander graph G of $n = 2^{65}$ nodes. With $n = 2^{65}$, each vertex of the Gabber-Galil expander graph of the form (x, y) can be represented using 64 bits with x and y being 32 bit each. The random numbers generated by our construction are the 64 bit vertex ids of the expander graph G.

We initialize our generator by having each thread start at a random vertex of G and performing an initial walk of length 64. To select the starting position, we need 64 random bits for each thread. In our implementation, we make use of the CPU for this as follows. These random bits are generated on the CPU and are supplied to the GPU. As current GPUs support asynchronous memory transfers, we can pipeline the execution and transfer. The details of this process are explained in Algorithm 6.

In Algorithm 6, we use labels such as `CPU`, `GPU`, and `CPU`$\rightarrow$ `GPU`. These represents the executions that are happening at the individual processors at any point of time. We employ these labels in order to distinguish between the parallel computations which are being carried out in each of these devices. The label `CPU`$\rightarrow$ `GPU` represents the asynchronous data transfer from the CPU to the GPU. The function $f(u, k)$ which is used in line 7 returns the k^{th} neighbor of u according to the definition of the Gabber-Galil expander graph.

Algorithm 6 InitializeGenerator(G, l, bin)

Input: A $7 - regular$ expander graph G of n nodes, the length of walk l and some random bits bin
Output: An initialized G

1: `CPU` : : Generate a random binary stream bin
2: `CPU`$\rightarrow$ `GPU` : : Transfer bin asynchronously
3: `GPU` : : **for** each node u in G do in parallel
4: `GPU` : : **for** i = 0 to l
5: $b(u)$ = $(int)(bin(t)$ & $(111 << (i * 3))$
{t is the Thread ID}
6: $v = f(u, b(u))$ { $f(u, b(u))$ gives the $b(u)^{th}$
7: neighbor of u}
8: $u := v$
9: **endfor**
10: **endfor**

In Algorithm 6, we first initialize a graph such that each of the node can represent a unique 64 bit number. As we have designed our algorithm to output 64 bit random numbers, each of the vertices u and v represent a unique (x, y) as described in the above section. For the initialization phase, we generate some random numbers using the CPU rand() utility which in turn calls the LCG present in the `glibc` library. As each of the vertices have 7 neighbors to identify from randomly, it requires only 3 bits to do so. The CPU streams random bits to the GPU as long as the kernel is executing, and the GPU has a constant supply of random bits to perform the walk. The overlap between the CPU and GPU computation will be explained in Section 5.4.

Once each thread completes a random walk of length 64, each thread is now ready to generate random numbers. To generate each random number, each thread has to essentially perform another random walk. As these walks are completely independent, our approach allows for massive parallelism. This can be done on demand also, unlike other GPU-based generators such as Mersenne Twister [86].

Let a walk be presently at vertex v. To continue the walk, we need to select a neighbor of v in G uniformly at random. This therefore requires few random bits. As earlier, we use the CPU to generate these random bits and supply them to the GPU in an asynchronous manner. At the end of the walk, each thread outputs a 64 bit random number. The details of this approach are explained in Algorithm 2. The asynchronous transfer in Line 1 is same as that of Algorithm 6.

Threads in an application that requires randomness can call the *GetNextRand()* routine to obtain random numbers. The *GetNextRand()* routine is described in Algorithm 7. The application does not require to pre-specify the number of bits before executing its kernels. The application has to only initialize the random number generator as described in Algorithm 6 before using *GetNextRand()*.

Algorithm 7 GetNextRand(G, bin)

Input: The initialized graph G and some random bits bin
Output: A new random number R

```
1: CPU→ GPU : : Generate and transfer bin asynchronously
2:     GPU : : for i = 0 to l
3:     b(u) = (int)(bin(t) & (111 << (i * 3))
           {here t is the Thread ID}
4:         v = f(u, b(u)) {f(u, b(u)) gives the b(u)^th
5:                 neighbor of u}
6:     u := v
7:     endfor
8: Return v
```

5.4 Experimental Results

Our experimental platform is described in Chapter 3. In Section 5.4.1 we study the speed and quality of our generator compared to existing generators. In Section 5.4.2, we study the quality of the random numbers produced by our generator. In all cases, the experiments are conducted over repeated trails and the average values are reported.

5.4.1 Performance Analysis

We now compare the performance of our generator to presently available GPU based generators. Some of the fastest generators on the GPU are the Mersenne Twister [86], and the CURAND utility [97]. We have therefore compared against these generators. In this experiment, we study the time taken to produce a random stream of N numbers for a given N ranging from 5 M to 1000 M. The resulting run times are plotted in Figure 5.2. In Figure 5.2, the label "Hybrid Timing" refers to the timings obtained by our generator. The label "Mersenne Twister" refers to the generator based on [87]. These timings were obtained by running the example code which is provided by NVidia with the CUDA toolkit. The label "CURAND" refers to the generator based on [97] from the CUDA library. CURAND is also an on-demand variant which can generate numbers on the fly as requested by an application when it is used in its Device API mode. We have considered the Device API for comparison since an on demand supply of random numbers is supported only on this mode. As can be seen from Figure 5.2, the hybrid generator outperforms both the Mersenne Twister based generator and the CURAND generator by a factor of 2 in most cases.

Another aspect of a hybrid algorithm to study is the overall resource utilization. In our program, the main work units are (i) an initial source of random bits, FEED, (ii) the transfer time required to transfer these initial source of random bits, TRANSFER, and (iii) the generation of random numbers using random walks on an expander, GENERATE. We map the FEED work unit onto the CPU and the GENERATE work unit on the GPU. This is a natural mapping as GENERATE is massively parallel

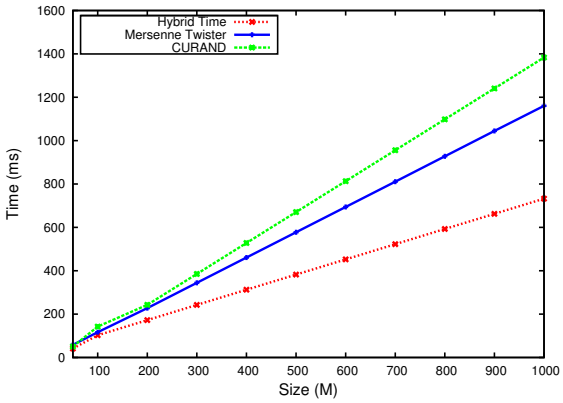

Figure 5.2 Timings across several list sizes

and hence can be done on the GPU. With this mapping, TRANSFER corresponds to transferring data between the CPU and the GPU using the PCI Express link. In Figure 5.3, the time taken for each of the above work units is indicated on the arrows. The arrow with label 6.2 ns corresponds to the TRANSFER work unit. As can be seen, the CPU is almost never idle, and the GPU is idle for about 20% during each iteration. The timings shown are for a batch size of 100 (see also Figure 5.4) where batch size, S, is defined as the number of random numbers each thread is generating. For instance, if N random numbers are to be generated, then with a block size of S, each of the N/S threads generate S random numbers each.

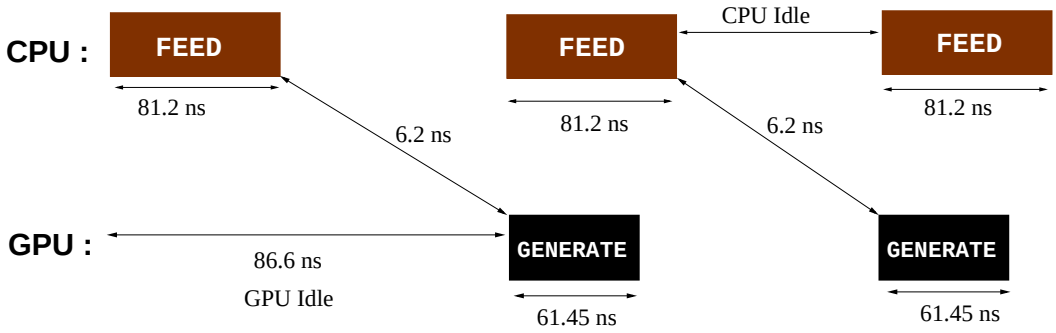

Figure 5.3 The overlapped execution of the work units.

In Figure 5.4, we study the variation of the timing with the block size. As we see, the timing is minimum at a work load of around 100 numbers per thread. This suggests that when the GPU threads are high but the work load per thread is low then the utilization of the system is low and the CPU stays idle for most of the time. Beyond 100 numbers per thread, the utilization is high but the CPU gets overloaded and the GPU starts to wait for CPU to transfer random bits. So, the time taken increases.

Comparison with `rand()`: Our hybrid generator can also work on other multicore architectures with minor programmatic changes. This is showcased by developing our generator for a multicore CPU

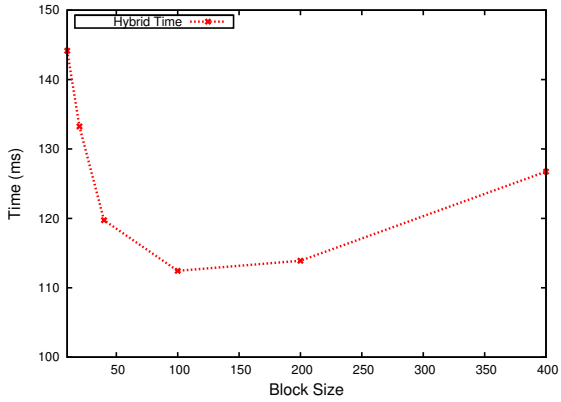

Figure 5.4 Variation of timing with block size.

alone. In this case, each core of the CPU runs threads which perform random walks on the implicitly defined expander graph.

On the CPU described in Chapter 3, we implemented our generator using the OpenMP specification 3.0 library. We then compare the time taken by our generator to that of the standard `glibc rand()` which is provided by the Fedora 14 Linux distribution. The result of this experiment is shown in Figure 5.5. The label "CPU Rand Time" refers to the time taken by `rand()` to generate the required quantity of random numbers. As can be seen, our generator scales up well compared to `rand()`. Further, our algorithm is thread safe.

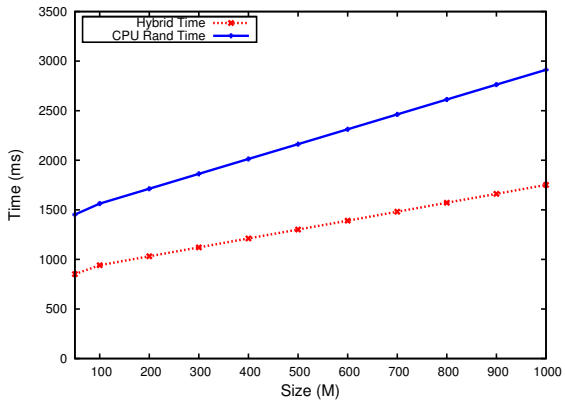

Figure 5.5 The time comparison when the algorithm runs on the CPU vs CPU `rand()`

Table 5.2 Quality Results of different algorithms

Algorithm	*DIEHARD* Tests Passed	KS-Test D
Hybrid PRNG	15/15	0.167
CUDPP_RAND	15/15	0.202
M. Twister	15/15	0.166
CURAND	8/15	0.534
`glibc rand()`	6/15	0.621

5.4.2 Quality

For studying the quality of our PRNG, we use several industry standard statistical tests. For instance, we have taken the *DIEHARD* battery of tests based on statistical testing methods as proposed in [31]. This suite consists of 15 different statistical tests. Marsaglia [83] implemented these tests so that these tests can be run for any PRNG easily. Each of the tests produces a p-value which is a measure of the uniformity in distribution of these numbers. Each of the p-values are further verified using Kolmogorov-Smirnov(KS) Test [31]. The KS test gives a measure of the uniformity of the numbers that are generated and how well they pass the *DIEHARD* tests.

The *DIEHARD* tests state that if the values generated by a PRNG are truly random, then the values are indistinguishable from a set of uniformly distributed values in the range of $[0, 1)$. The test statistic p should lie between 0.01 and 0.99 to pass the test. After the *DIEHARD* tests are completed, we also looked at the KS test against a set of uniformly generated numbers. This test tries to measure the maximal deviation between two curves drawn on a cumulative distribution function. A low value indicates the lower deviation from a set of uniformly distributed values. The test statistic D gives a measure of how well the generator performs in comparison to the other generators. As we see from Table 5.2, the KS test result of our algorithm is comparable to that of Mersenne Twister and better than that of CURAND.

Apart from the *DIEHARD* suite, we also use the *TestU01* suite. Pierre L'Ecuyer and Richard Simard implemented the *TestU01* software library [78] which is a more advanced test than the *DIEHARD* tests. This suite contains three different batteries: SmallCrush, Crush and BigCrush. These three tests are all in the increasing order of quality. The results obtained from the tests are tabulated in table 5.3.

As we see from Table 5.3, our generator passes all the tests in the SmallCrush battery. It is able to pass only fourteen and thirteen tests in Crush and BigCrush tests respectively. This is comparable to other presently available PRNGs on GPUs such as CURAND and the Mersenne Twister based generator.

Table 5.3 TestU01 test results

PRNG	Test Suite	Tests Passed
CURAND	SmallCrush	15/15
	Crush	14/15
	BigCrush	13/15
M.Twister	SmallCrush	15/15
	Crush	13/15
	BigCrush	13/15
Hybrid PRNG	SmallCrush	15/15
	Crush	14/15
	BigCrush	13/15

5.4.3 Discussion

One of the interesting aspects of our implementation is the use of CPU generated random numbers to aid the GPU based generator. This can be justified by the following arguments regarding quality and performance.

Our generator based on random walks on expander graphs, produces quality pseudo random numbers as can be seen from the results of Section 5.4.2. The quality of our generator is better than `glibc rand()`, and is comparable or better compared to existing PRNGs on GPUs. Hence, our technique can be seen an improving the quality of a naive random number generator. Further, this increase in quality is obtained by using a little amount of initial randomness. Our technique has connections to other works on expander graphs such as probability amplification [90].

To perform a random walk, one has to select a neighbor of the current position of the walk uniformly at random. This requires some source of randomness. Using a GPU based generator for this purpose is not a good solution as it would still keep the CPU idle. Further, there are no fast, on-demand GPU based generators. Hence, we make of the CPU to provide us with these few random bits that are used by the GPU. This is also helping us improve the performance of our generator. Our generator is the fastest known PRNG on GPUs as can be seen from Figure 5.5.

In our framework, we notice that the GPU is idle for a small fraction of the time. We are presently using the ANSI C based `rand()` for this purpose. This is mainly due to the fact that there is no fast randomness generator on a multicore CPU. Our generator, working on a multicore CPU can be used in the place of `rand()`. As Figure 5.5 shows, this would help us in never keeping the GPU idle.

5.5 Application : Hybrid Monte Carlo

Monte Carlo methods are used in several areas of science to simulate complex processes, to validate simpler processes, and to evaluate data. In Monte Carlo (MC) methods, a stochastic model is constructed

in which the expected value of a certain random variable is equivalent to the physical quantity to be determined. The expected value of this random variable is then determined by the average of many independent samples representing the random variable. These independent samples are constructed by the use of random numbers following the distribution of that random variable.

Photon migration, i.e., light propagation in a random media, is an area where MC simulations are proven as a gold standard [21]. In this method, several photons are launched with their position and direction initialized to either zeros (for some pencil beam initialized at the origin) or some random numbers. A variance model is used for simulation, where the absorption of photons is simulated by reducing weights and not discrete termination. At every step a photon takes, a fraction of its weight is absorbed, and then photon packet is scattered. The new direction and weight of photon are updated. After several such steps if the remaining weight of a photon is below a certain threshold, the photon is terminated.

To summarize, the rules of photon migration can be expressed as step-size of photon movement between sites of photon tissue interaction, and the angles of deflection in the photons trajectory in case of a scattering event. The method is statistical in nature and we need to study the propagation of a large number of photons. Due to this, the method requires a large amount of computation time. An earlier work of photon migration on GPU [7] does this simulation across multiple layers of absorption. This work uses a multiply with carry (MWC) based RNG to initialize the weights of the photons. We use the same implementation to show that the simulation can happen in a much better way when it is plugged in with the hybrid PRNG.

5.5.1 Our solution

We try to solve the problem of photon migration using the hybrid PRNG which has been explained in the previous sections. The hybrid PRNG supply the random numbers which are used to initialize all the simulation kernels used. There are specifically three simulation kernels which are used to simulate the three different layers of the MC simulation. Our hybrid PRNG supplies the values which are required at each layer.

The PRNG is the heart of the multi-layer simulation of photons. Each of the initial weights of the photons must be set at a random value which should be generated independent of each other in order to minimize the number of weight clashes that might happen at the different layers. These clashes correspond to certain atomic operations. The better quality of random numbers ensure that there will be lesser clashes and hence lesser serialization will occur. The hybrid PRNG works completely in a data parallel way which ensures that whenever a call for the PRNG is made, it supplies a random number irrespective of that at the other thread calls. The algorithm proceeds in an iterative fashion where a fixed quantity of photon packets are processed in each iteration. This provides an ideal setting for the PRNG to work as the GPU kernel execution times can be used to supply the graph with fresh random bits which shall be used for random walks in subsequent iterations. Also, the high quality of the random numbers supplied allows for more number of photons to be simulated at each layer.

Algorithm 8 $MCPhotonMigration(P, LayerParams)$

Input: Number of photons P and parameters of layers
Output: Reflectance parameters and absorbed fraction

1: Initialize 7-regular graph G by Algorithm 6
2: `CPU` **::** Generate and transfer asynchronously random binary stream bin.
3: `GPU` **::** Launch a photon after initializing weights with getnextRand()
4: **While** the photon survives
5: `GPU` **::** Remove the absorbed weight
6: `GPU` **::** Scatter the photon
7: `CPU` **::** Generate and transfer new bin in an
8: overlapping manner
9: `GPU` **::** $noOfUsedPhotons+ = 1$
10: **If** $noOfUsedPhotons \leq maxNoOfPhotons$
11: Goto step 3.
12: **end while**

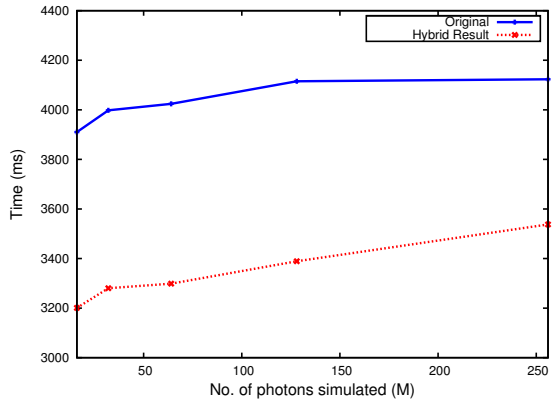

Figure 5.6 Variation of timing with number of photons simulated

In Algorithm 8, we see the pseudo code of the algorithm applied. The PRNG is used in the algorithm by using an overlap between the CPU and the GPU for generating and transferring the required random bits. As the generation is not related to the GPU kernels, we have optimally made the CPU work towards re-populating the bin array while the GPU is busy in steps 5 and 6. The result of the experimentation can be observed from Figure 5.6. The number of photons is varied from 1 M to 256 M. The Y-axis shows the time taken by our method, labeled 'HybridResult', and the time taken by the implementation of [7], labeled 'Original'. We can attribute this result to the following reasons:

- **Reduced memory transaction overhead:** As the PRNG works in an hybrid fashion, the actual memory overhead of accessing the global memory for getting random numbers is minimized. This is due to the reason that our implementation does not use any extra space for storing the random numbers unlike [7]. In our model, random numbers are generated on the fly. Hence a certain speedup is obtained.

- **Lesser number of clashes in atomic operations:** The quality of the hybrid PRNG has been already discussed in Section 5.4.2. This is another advantage which the hybrid PRNG offers to the MC simulation. As all the threads independently supply high quality random numbers to initialize the weight of photons, the number of clashes happening is subsequent layers is reduced. By clashes, we mean the behavior of two photons as a single one due to having the same weight. As a higher amount of photons have unique weights, they are independently simulated. The weights of these photons quickly fall below the threshold and are terminated. As a result, if we are aiming to simulate a fixed number of photons, then all of them are simulated at a much lesser amount of time. This contributes towards an overall speedup of around 20%.

5.6 Conclusions

In this work, we have presented an efficient pseudo random generator for GPUs. Our generator satisfies all the conditions that are deemed important. Our generator also combines the computational abilities of multicore CPUs and GPUs in a clever way to improve resource utilization. In future, we wish to study cryptographic applications that often require high quality random numbers.

PART II

Graph Algorithms

Introduction

Graphics processing units provide a large computational power at a very low price which position them as an ubiquitous accelerator. General purpose programming on the graphics processing units (GPGPU) is best suited for regular data parallel algorithms. They are not directly amenable for algorithms which have irregular data access patterns such as list ranking, and finding the connected components of a graph, and the like.

Parallel computing on graphs however is often very challenging because of their irregular nature of memory accesses. This irregular nature of memory access also stresses the I/O systems of most modern parallel architectures. It is therefore not surprising that most of the recent progress in scalable parallel graph algorithms is aimed at addressing these challenges via innovative use of data structures, memory layouts, and SIMD optimizations [91, 49, 112]. Recent results have been able to make efficient use of modern parallel architectures such as the Cell BE [112], GPUs [91, 58, 49], Intel multi-core architectures [25, 142, 3] and the like. Algorithms running of GPUs have shown standout performance amongst these because of its massive parallelism.

The architecture of GPUs the data parallel computing model best where a common processing kernel acts on a large data set. Several general purpose data parallel applications [96] and higher-order primitives such as parallel prefix sum (scan), reduction, and sorting [34, 116], [23] have been developed on the GPU in recent years. From all these applications, it can be observed that GPUs are more suited for applications that have a high arithmetic intensity and regular data access patterns. However, there are several important classes of applications which have either a low arithmetic intensity, or irregular data access patterns, or both. Examples include list ranking, an important primitive for parallel computing [76], histogram generation, and several graph algorithms [49, 137, 136]. The suitability of GPUs for such applications is still a subject of study. Very recently, the list ranking problem on GPUs is addressed in [76] where a speed up of 10 is reported with respect to the CPU for a list of 64 million nodes.

In lieu of the above reasons we explore the graph algorithms in this part of the thesis and explain our solutions in List Ranking, Graph Connected Components, and Breadth First Search. List Ranking is discussed in Chapter 6 where we show a list shortening based approach for traversal and ranking. In Chapter 7, we discuss Hybrid Graph Connected Components. Breadth First Search for Hybrid parallel machines is discussed in Chapter 8.

Chapter 6

Parallel List Ranking

6.1 Introduction

In many previous works it has been shown that GPUs are good at improving the performance of regular computations such as those described in [116, 46, 27]. On such regular applications, GPUs can outperform a single-core CPU performance by a large factor on average. In recent times, researchers have studied how GPUs perform on irregular computations such as list ranking [139, 108], connected components [123], among others. It is to be noted that in these cases, the speed-up compared to a single core CPU performance is only of the order of 10 or less. More recently, in [79], it is also argued that on a broad class of throughput oriented applications, the GPGPU advantage is only of the order of 3x on average and 15x at best compared to the best known CPU implementations.

In this work, we explore the efficiency of a hybrid computing platform, consisting of a CPU and a GPU, on two classical irregular computation based problems: list ranking, and graph connected components. Our hybrid implementations for these problems improve on corresponding GPU based solutions by a factor of 25% to 30% on an average. In arriving at our algorithms and implementations, we bring several analytical and empirical issues in hybrid algorithms to the fore. We also discuss issues such as scalability, resource utilization, and synchronization in hybrid computing platforms.

6.1.1 Related Work

General purpose computing on the GPUs, termed GPGPU, has matured enough in the research community. Some of the important works include [46, 123, 139]. In most of these works, the computation is entirely performed on the GPU. In this chapter, we do not wish to provide a complete review of these works.

In a recent work, Lee et al. [79] reported that typical GPU performance is on the average about 3 times faster than a multicore CPU performance on a class of throughput centric workloads. The main message from [79] is that the CPUs are evolving and are matching the performance of GPUs up to a small constant. It therefore makes practical sense to use both the CPU and the GPU for computation.

There are very few hybrid works reported in the literature so far. An early work is that of Tomov, Dongarra, and Baboulin [126]. In [126], the authors use a combination of GPU and CPU to solve dense linear algebra problems. Other recent works in this direction include hybrid approaches for QR factorization [6], Cholesky factorization [82], triangular forms [130], and other associated works [128, 129, 127]. These works justify the applicability of hybrid computing platform for a variety of dense linear algebra problems.

In [50], the authors utilize a CPU and GPU combination to give a parallel solution for the push-relabel maximum flow algorithm. Their work optimally uses the two devices via a switching mechanism to perform the push and the relabel operations on the two devices respectively. However, the work does not focus on a overlapped execution between the two devices. Either of the devices perform some work while the other is idle. Our work tries to use as much of overlap as possible in order to extract a higher amount of data parallelism. This intra-device parallelism also ensures a higher amount of resource utilization and compute power which is available in a single setup. In [139] the authors show how to perform efficient list ranking on GPUs. They use the CPU to perform a part of the computation, during which the GPU stays idle.

There has been recent work on list ranking on new and emerging architectures such as the IBM Cell and GPUs, apart from algorithmic solutions in the PRAM model. In the parallel setting, several optimal solutions have been proposed which uses the sparse-ruling sets like the one by Helman-JaJa [52]. The Helman-JaJa solution has been used in the work of [11] on the Cell, in the work of [108] and [139] on the GPUs. At present, the results of [139] outperform all the earlier works.

We now focus briefly on the literature with respect to finding the connected components of a given graph. There are several PRAM algorithms for this problem, e.g., [55, 119]. The PRAM model is not perfectly suitable for GPU computing, as it does not account for several factors such as memory latency and hierarchy, communication latency, and synchronization, among others. In [48], the author presented many optimizations for the PRAM model along with results in several multi-processor systems.A GPU optimized version of the SV algorithm has been studied by Soman et al. in [123].

6.1.2 Our Results

In this chapter, we use the hybrid computing platform as described in Chapter 2. On this platform, we study two fundamental problems: list ranking and graph connected components. For these two problems, we employ different kinds of hybrid parallelism.

We design an efficient hybrid list ranking problem that uses techniques such as fractional independent sets, and the Helman-JaJa algorithm[52]. Our algorithm can be seen as adding a preprocessing phase to the Helman-JaJa algorithm that may be of independent interest even in a non-hybrid setting. Our algorithm for list ranking is then implemented on a CPU+GPU hybrid platform. (See Section 2 for a description of our computing platform). Our implementation can rank a list of 128 M nodes in about 287 ms which is faster by 50% compared to the fastest reported algorithm for list ranking [139]. The gains

we obtain can be attributed to the hybrid problem solution. Also, we use the principles of producer-consumer problems to minimize the idle times.

Additionally, we show that efficient hybrid algorithms can be designed for finding the connected components of a graph. Our hybrid algorithm uses a variant of the popular Shiloach-Vishkin parallel algorithm [119], and the sequential depth first search algorithm. Our implementation on a CPU+GPU hybrid platform achieves an average speed-up of 25% compared to the best possible GPU implementation [123]. We also notice that our hybrid algorithm has very minimal idle time. We also show that our approach can lead to auto-tuning.

6.2 Recursive Hellman Jaja Algorithm for List Ranking

The algorithm, as described by Helman and JaJa (RHJ) [52] reduces a list of size n to s and then uses one processing node to calculate the prefix of each node of this new list. It would be unwise to apply the same technique to the GPU, as this will leave most of the hardware underutilized after the first step. Instead, we modify the ranking step of the original algorithm to reduce the list recursively until we reach a list that is small enough to be tackled efficiently by the GPU or by handing over to either the CPU for sequential processing or to Wylies algorithm on the GPU. Each element of an array L contains both a successor and rank field. In first step of the algorithm, we select p splitter elements from L. These p elements will denote the start of a new sublist and each sublist will be represented by a single element in L_1 . The record of splitters is kept in the newly created array L_1 . Once all the splitters are marked, each sublist is traversed sequentially by a processor from its assigned splitter until another sublist is encountered (as marked in the previous step). During traversal, the local ranks of the elements are written in list R. Once sublist has been traversed, the successor information, i.e the element (with respect to indices in L_1) that comes next is recorded in the successor field of L_1 . It also writes the sum of local ranks calculated during the sublist traversal to the rank field of the succeeding element in L_1 .

Next step performs the recursive step. It performs the same operations on the reduced list L_1 . First step of the algorithm determines when we have a list that is small enough to be ranked sequentially, which is when the recursive call stops.

Finally, we add the prefix values of the elements in R_1 to the corresponding elements in R. Here R_1 contains all the sublist lengths. Upon completion of an iteration of RHJ, the List R will have the correct rank information with respect to the level in the recursive execution of the program.

Implementing the RHJ algorithm on the GPU is challenging for the following reasons: Programs executed on the GPU are written as CUDA kernels. They have their own address space on the GPUs instruction memory, which is not user-accessible. Hence CUDA does not support function recursion. Also, each step of the algorithm requires complete synchronization among all the threads before it can proceed.

6.2.1 CUDA implementation of RHJ

All operations that are to be completed independently are arranged in kernels. Global synchronization among threads can be guaranteed only at kernel boundaries in CUDA. This also ensures that all the data in the global memory is updated before the next kernel is launched. Since each of the 4 phases of the algorithm require a global synchronization across all threads (and not just of those within a block) we implement each phase of the algorithm as a separate CUDA kernel. Since we are relying purely on global memory for our computation, CUDA blocks do not have any special significance here except for thread management and scheduling.

The first kernel is launched with p threads to select p splitters and write to the new list L_1. The second kernel also launches p threads, each of which traverse its assigned sublist sequentially, whilst updating the local ranks of each element and finally writing the sublist rank in L_1. The recursive step, is implemented as the next iteration of these kernels, with the CPU doing the book-keeping between recursive levels. Finally we launch a thread for each element in L to add the local rank with the appropriate global sublist rank.

A wrapper function that calls these GPU kernels is created on the CPU. It takes care of the recursive step and the recursion stack is maintained on the host memory. CUDA also requires that all the GPU global memory be allocated and all input data required by the kernels be copied to the GPU beforehand. Once we obtain the list size, the required memory image is created for the entire depth of recursion before we enter the function.

The final sequential ranking step (which is determined by the variable limit) can be achieved in a number of ways. It can either be handed over to Wyllies algorithm, or be copied to the CPU or be done by a single CUDA thread (provided that the list is small enough). An appropriate value of limit for each of these scenarios is discussed in the results section.

6.3 Hybrid List Ranking

The problem of list ranking can be stated as follows. Given a linked list of n nodes, find the distance of each node in the list from one end of the list. The problem is easy to solve in the sequential setting with a linear time solution. The problem is difficult to solve in the parallel setting as the solutions are quite non-trivial and differ significantly from known sequential algorithms. However, list ranking is identified as one of the fundamental problems in parallel computing by Wyllie [141] in 1978. Since then, there have been several solutions to this problem. The range of techniques employed in existing PRAM algorithmic solutions include independent sets, ruling sets, and deterministic symmetry breaking [8, 141, 52, 120, 109].

Given the importance of the problem, recently solutions optimized to emerging architectures such as the GPU [108, 139], and the IBM Cell [11] have been reported. The solution of [108] uses only the GPU to perform list ranking. The solution of [139] uses the CPU also but not in the manner we envisage in hybrid multicore computing. In [139], ranking a list of small number of splitters is done on the CPU

followed by the global ranking on the GPU. Notice that while the CPU is working on the solution, the GPU stays idle, and vice-versa.

In this section, we present a hybrid multicore algorithm to the list ranking problem using ingredients such as fractional independent sets, and the Helman-JaJa algorithm. Our algorithm can be seen as adding a preprocessing phase to the Helman-JaJa algorithm that may be of independent interest even in a non-hybrid setting. In the hybrid setting, our solution can be seen as a pipelined parallelism based solution as we use the approach of pipelined parallelism while still being a hybrid solution where both the CPU and the GPU are working simultaneously for a majority of the time.

6.3.1 The Proposed Solution

Our list ranking algorithm can be described as follows. The basic idea is to remove a lot of nodes from the given linked list L so that we can rank a small list L' quickly. Once L' is ranked, we can reinsert the nodes removed and obtain the ranks for every node in L. The pseudo-code for our approach is shown in Algorithm 9.

Algorithm 9 $Listrank(List\ L)$

1: Shrink L to a small list L' by removing nodes from L
2: Rank the list L'
3: Re-insert removed nodes from L into L' and rank L

To be able to do Step 1 efficiently, we need a fast mechanism to remove lot of nodes from L in one iteration. One of the mechanisms to achieve this is to choose a maximal independent set of nodes that can be removed from L. However, to be time-optimal, this requires $O(\log n)$ time per iteration [64]. However, linked lists are a class of graphs which have the property that a fractional independent set can be computed also in parallel efficiently. Hence, we use this approach and also show below how to compute a fractional independent set efficiently.

6.3.2 Fractional Independent Sets

Given a graph $G = (V, E)$, a (c, d)–fractional independent set (FIS) is an independent set of nodes $U \subseteq V$ so that:

- $|U| \geq |V|/c$, for some constant c,
- the degree of any node $u \in U$ is at most d, for a constant d.

For a linked list L of n nodes, to compute a fractional independent set in parallel, we proceed as follows. Each node v picks a bit, $b(v) \in \{0, 1\}$, uniformly at random and independent of other nodes. Then, we say that a node v belongs to the FIS if $b(v) = 1$ and neither the predecessor nor the successor of v also chose 1. It can be seen from relatively simple arguments (cf. [64]) that with high probability, the FIS constructed above has at least n/c nodes for $c \geq 24$.

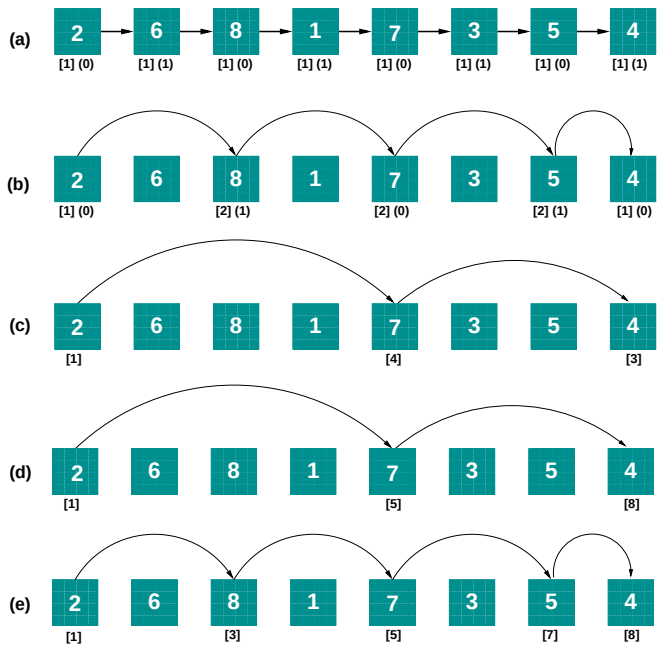

Figure 6.1 The linked list and pre-processing done. (a) The initial list with the rank values in square brackets and random string in parenthesis. (b) Elements removed on the basis of the random string and ranks adjusted (c) List obtained after the pre-processing, (d) Ranking the remaining list, and (e) Restoration of the nodes removed from the list.

Our complete algorithm is shown in Algorithm 10. An example run of the algorithm is shown in Figure 6.1. We discuss each phase of the algorithm in detail as follows.

Phase I Issues: Recent works on list ranking [108, 139, 11], that involve sublist ranking spend the maximum amount of time on sublist ranking. Hence, it would be interesting to see how this can be minimized. Our preprocessing technique helps in that direction as the list size reduces by a nonlinear factor in a small number of iterations. In addition, spreading this computation on both the CPU and the GPU in a pipelined manner helps us reduce the overall time taken further.

In theory, it holds that $O(\log \log n)$ iterations of Phase I suffice to reduce the size of the remaining list to $O(n/\log n)$ (cf. [64]). In practice, we need that the list size reduces to an extent so that the overall time taken by Phases I and II is minimized. We explore this trade-off experimentally in Section 6.4.1. The number of iterations r will be discussed later in Section 6.4.1.

In Algorithm 10, we note that in Phase I generating random numbers and using random numbers to find nodes that would be part of the FIS and will be removed from the list can be processed in a pipelined manner. See also Figure 6.4. The technique is similar to that of double buffering, where the CPU generates random numbers and the GPU uses these random numbers. While the GPU is computing using one set of random numbers, the CPU populates another set of random numbers to be used in the next iteration. The choice of generating random numbers on the CPU and using them in lines 5–11 of

Algorithm 10 $ListRank(List\ L)$

Input: A list of size n
Output: Ranks of the list provided

Phase I : *Pre-processing* CPU ::
for r iterations in parallel **do**
    `CPU` **::** Generate a random binary stream bin
    `CPU`$\rightarrow$ `GPU` **::** Transfer bin asynchronously
    `GPU` **::** **for** each node u in the list do in parallel
        Let $b(u)$ be the bit choice of node u
        if $b(u) = 1$ and $b(pred(u)) = 0$ and
        $b(succ(u)) = 0$ **then**
            Remove node u with proper book-keeping
        endif
    endfor
end for
Phase II : *Sublist Ranking on the GPU*
`CPU` **::** Generate a random binary stream
`CPU` $\rightarrow$ `GPU` **::** Transfer random binary stream to the GPU
`GPU` **::** Select random splitters in the list obtained after Phase I
Rank each sublist locally in parallel on the GPU
Find global ranks of splitters
Compute global ranks of all elements
Phase III : *Re-insertion*
`GPU` **::** Re-insert, and hence rank, the elements removed in Phase I

Algorithm 10 is justified in the following section. Lines 5–11 of Algorithm 10 present a highly data parallel program that is more suitable to be executed on the GPU.

Phase II Issues: In Phase II, the manner in which the remaining list at the end of Phase I is stored makes it difficult to choose splitters. For instance, if we pick splitters at regular intervals as is done in [108], we may choose entries in the successor array S that are no longer part of the list. A similar problem arises when we use the technique of [139]. For this reason, we perform an element compaction from S to S' where S' contains only the non-removed elements of S. One can then pick splitters in S' and use them as splitters for the remaining list.

Phase III Issues: Phase III is the opposite of Phase I, and the book-keeping done in Phase I is useful to insert the elements in the right order. Our algorithm follows the model of most parallel list ranking algorithms [64].

6.4 Computing FIS using PRNG from Chapter 5

We now show the working of our algorithm using the PRNG that we defined in Chapter 5. In this algorithm, we call the getNextRand() function which was defined in Algorithm 7 in Chapter 5.

Algorithm 11 $ReduceList(\ L, G)$

Input: A list L of size n and a 7 regular graph G
Output: A sublist of $n/\log n$ nodes

```
1:  Phase I : Pre-processing
2:  CPU :: Initialize 7-regular graph G by Algorithm 6
3:  for r iterations in parallel do
4:      CPU : : Generate and transfer asynchronously random binary stream bin
5:      GPU : : for each node u in the list do in parallel
6:          temp=getNextRand(bin)
7:          Let b(u) be the bit choice of node u from temp
8:          if b(u) = 1 and b(pred(u)) = 0 and
9:          b(succ(u)) = 0 then
10:             Remove node u with proper book-keeping
11:         endif
12:     endfor
13: end for
```

We now describe Algorithm 11 in more detail. We first initialize a 7-regular Gabber-Galil expander graph. This graph is then used to generate random numbers as required. As can be seen in Line 6 of Algorithm 11, each thread can call `getNextRand()` to obtain a new random number that will be used by this thread. Since this operation is done only for those nodes in L that are not removed in previous iterations, the number of such calls is not known apriori. One can only say that the number of nodes in L reduces by a constant factor in each iteration. Further, the GPU compute time is overlapped with an asynchronous transfer from CPU.

The ability to efficiently produce random numbers on demand in our current approach offers a big advantage to our implementation compared to the hybrid implementation of Algorithm 10. In Algorithm 10, the CPU generates a quantity of random numbers that is predetermined to be an upper bound on the number of nodes remaining in the list at each iteration. We will show shortly in our results that an on demand generation reduces the runtime by 40%.

Once we have a list of size $n/\log n$, we continue with the approach of Phases II and III of the algorithm in [16].

6.4.1 Experimental Results

The experimental platform which we use has been already described in Chapter 2. The list is stored in a structure with the successor, predecessor, and the rank values. We experimented only on random lists as they are the most difficult to rank due to their irregular nature of memory accesses.

In our experiments, we vary the size of the lists and measure the total running time. We compare the time taken by the algorithm presented in Algorithm 10 with the following alternative approaches. The algorithm presented in [139] is presently the fastest known algorithm for list ranking. We therefore pick this algorithm and show that our hybrid approach is an improvement over the algorithm of [139]. We also run the algorithm presented in Algorithm 10 in a non-hybrid manner by moving all the work to the GPU. We call this a *pure GPU* algorithm. Here two possibilities arise as there are two known methods to generate random numbers on the GPU. The CUDPP library function `cudppRand()` uses the MD5 based algorithm from [131]. A recent work [105] uses the Mersenne Twister based approach to generate random numbers on the GPU. Both these two methods are used for experimenting on Algorithm 10.

Figure 6.2 shows the runtime of our algorithm on lists of various sizes in comparison with the timings from Wei and JaJa [139]. In Figure 6.2, the label "Hybrid time" refers to our approach, and the label "WJ10" refers to the timings from [139]. The label "Pure GPU-MD5" refers to the pure GPU algorithm that uses the CUDPP library function `cudppRand()` to generate random numbers. The label "Pure GPU-MT" refers to the pure GPU algorithm that uses the Mersenne Twister based random number generation [105].

It can be observed from Figure 6.2 that the improved runtime of our approach is due to both algorithmic improvements and also due to an efficient use of the computing platform. For instance, adding the preprocessing step, Phase I, to the algorithm of [139] is an algorithmic improvement that reduces the time for list ranking as can be seen from the plots labeled "Pure GPU-MD5" and Pure GPU-MT". Similarly, running Algorithm 10 on the GPU alone, by generating random numbers also on the GPU, improves the results of [139] but is still slower compared to the hybrid algorithm. This figure also shows that the scalability of our approach does not suffer. We were able to run the experiments on lists of sizes upto 128M.

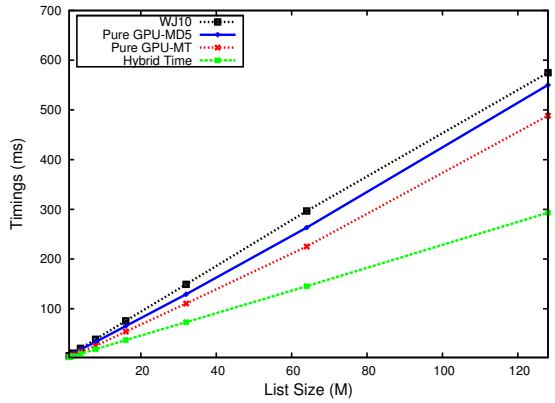

Figure 6.2 Time Comparison with respect to pure GPU implementations and the best known result.

In the plot labeled "WJ10", we use the numbers reported in [139]. Since the GPU hardware used by [139] and the GPU in our computing platform are of the same generation and make, the comparison is appropriate. We could not access the code from the authors of [139]. In the other three plots, we use the same datasets and run the corresponding programs over multiple runs and considered the average runtime in obtaining the plots. The results have a low variance.

In our work, to generate random numbers on the CPU, we use the standard library routine `rand()` which is implemented in the glibc library of most linux distributions. The glibc `rand()` call implements a linear congruential generator to produce random numbers.

We now study the timing trade offs between Phase I and Phase II. To this end, we vary the number of iterations. r in line 2 of Algorithm 10, and measure the time taken by Phase I and Phase II. On extensive experimentation we observed that a value of r between $4 \log \log n$ and $5 \log \log n$ gives the best possible overall time across several values of n. This is illustrated further by Figure 6.3 for a list of 128M. We measure the time taken by Phase I, Phase II, and the total time. (Since Phase III consumes very little time, its effect on this trade off is ignored.) The time taken by Phase I increases as we increase r, and this has the effect of reducing the size of the remaining list. Therefore, the time taken by Phase II decreases as r increases. The total time, which is very close to the time taken by Phase I plus the time taken by Phase II has a minimum at $r = 4 \log \log n$. In the plot labeled "Hybrid time" in Figure 6.2, the values reported correspond to the above value of r.

Table 6.1 Execution times across several list sizes. The list sizes are in million and the timings in milliseconds

List Size	Phase I	Phase II	Phase III	First Transfer	Total	GPU Time [WJ10]
1	1.696	0.721	0.096	0.114	2.636	5.593
2	3.293	1.397	0.095	0.207	4.992	10.711
4	6.346	2.757	0.092	0.395	9.591	20.342
8	12.525	5.352	0.096	0.772	18.745	38.472
16	25.452	10.565	0.096	1.422	37.535	75.691
32	49.423	20.884	0.111	2.661	73.079	140.182
64	98.756	41.686	0.113	5.213	145.763	296.723
128	196.512	83.015	0.114	7.312	286.953	574.911

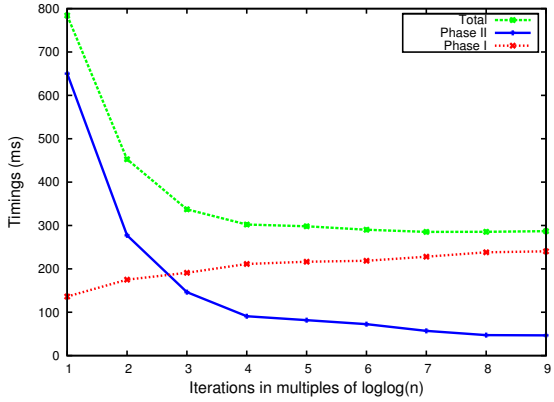

Figure 6.3 Trade offs between Phase I and Phase II and the total timings for a 128M sized list

In Table 6.1 we show the running time of each phases of our algorithm for lists of various sizes. It can be noticed that Phase I dominates the overall time taken. Nevertheless, our total time consumed is about 50% lower compared to [139]. Phase III, while running for the same number of iterations as Phase I, still consumes very little time. This is because of the nature of computation in Phases I and III. Phase I has lot more irregular memory accesses compared to that of Phase III, hence the difference in their time consumption.

Since the present hybrid algorithm outperforms the results of [139], these results will also improve the corresponding results from [108, 11, 109].

Table 6.2 Generation, transfer and pre-processing timings for a list of 128 million. All the timings are in Milliseconds. Iteration 0 refers to the generation of the first batch of random numbers on the CPU and their transfer.

Iteration	Generation Time	Transfer Time	Preprocessing Time
0	32.40	0.163	—
1	28.35	0.096	30.45
2	22.59	0.082	25.08
3	18.70	0.053	20.68
4	15.25	0.040	18.35
5	12.88	0.035	14.08
6	10.56	0.027	12.84
7	8.30	0.013	10.66
8	6.09	0.008	8.50

To showcase the power of our hybrid algorithm approach, we focus on Phase I of our algorithm and illustrate the runtime of various steps in Phase I. The main steps in Phase I are generating random numbers on the CPU, transferring random numbers to the GPU, and using random numbers on the GPU to find and use an FIS. We instrument our program to profile the first 9 iterations of Phase I, and the results are shown in Table 6.2. The first row denotes the first generation and transfer. It can be noticed from Table 6.2 that the time taken by the GPU to use the random numbers and remove nodes from the linked list is greater than the time required to generate the random numbers on the CPU and transfer them to the GPU. This is achieved by using an asynchronous transfer technique wherein a transfer from the CPU to the GPU can be overlapped with GPU execution.

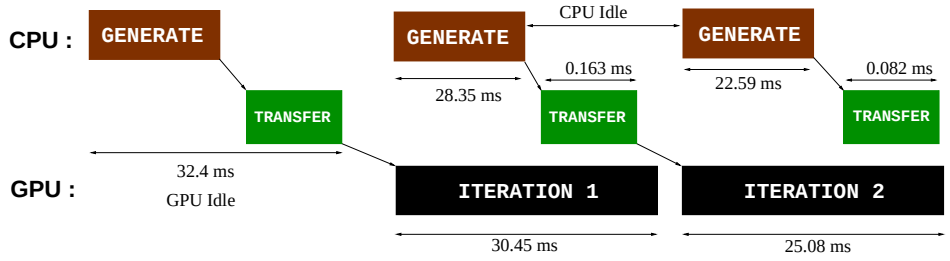

Figure 6.4 Work units in overlapped execution for a list of 128 M elements for the first 2 iterations.

Table 6.3 Generation, transfer and pre-processing timings for a list of 16 million

Iterations	Generation Time	Transfer Time	Preprocessing Time
1	4.35	0.057	5.45
2	3.59	0.040	3.08
3	2.70	0.033	2.68
4	2.25	0.030	2.35
5	1.88	0.025	2.08
6	1.56	0.017	1.84
7	1.30	0.013	1.66
8	1.09	0.008	1.50

The above scheme suggests that in our implementation, the GPU is never idle except for the first time when random numbers are being generated on the CPU. This situation is illustrated in Figure 6.4 where the time taken by each step of Phase I is shown for a list of 128 M elements. Figure 6.4 shows also how the various steps in Phase I overlap in their execution. It can be indeed noticed from Figure 6.4 that the GPU is idle only for less than 5% of the time.

6.4.1.1 Results using PRNG from Chapter 5

In our experiments we vary the list size to upto 128 million elements and compare the results with the result of Algorithm 10. We also compare our results with two other techniques where the glibc random number generator used in Algorithm 10 is replaced with a generator based on Mersenne Twister. This is referred in Figure 6.5 as "Pure GPU MT" and is hence a pure GPU implementation without any involvement of the CPU. To summarize our generator outperforms the fastest running algorithm by almost 40%.

It can be seen from Figure 6.5, that the improvement in runtime for reducing the list to a size of $n/\log n$ is due to two factors. Firstly, using hybrid algorithms helps by bringing also the CPUs into the computation process. Secondly, an efficient on demand PRNG helps in reducing the runtime further as can be seen from the plots labeled Hybrid-glibc and Hyrbid-PRNG. Given that Phases II and III of list ranking take only 20% of the overall time, using Algorithm 10 for Phase I of list ranking as described in [16] would result in an improvement of 50% in the runtime of list ranking over various list sizes ranging up to 128 M nodes.

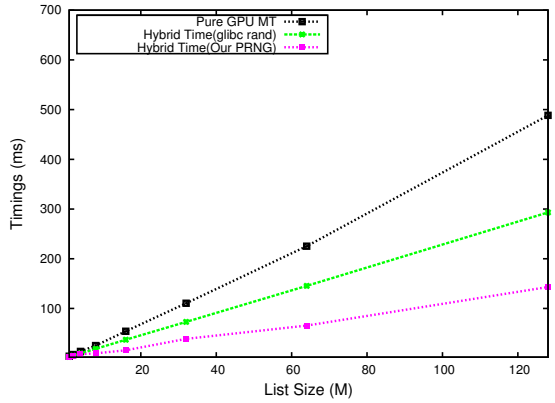

Figure 6.5 The timing comparison with the other algorithms.

6.5 Conclusions

Our work here considered two fundamental problems and proposed hybrid multicore algorithms for these two problems. Our work has brought up several important analytical issues with respect to hybrid algorithms.

In our present work, we considered a very simple hybrid computing platform with only one multicore CPU and a GPU. Our hybrid algorithms however are designed to scale when more CPUs or more GPUs are added to the computing platform by making appropriate changes.

As hybrid algorithms become popular, it is also important to consider new programming mechanisms that allow for implementing hybrid algorithms efficiently. For instance, mechanisms to improve the synchronization support across devices can help programmability of hybrid algorithms.

In future, we wish to consider further fundamental problems for which one can design hybrid algorithms. Other analytical issues that can be considered are: a mechanism to arrive at an optimal assignment of tasks to processors, a notion of critical path in such an assignment, and the like.

Chapter 7

Hybrid Graph Connected Components

7.1 Introduction

Graphs are an important data structure in Computer Science because of their ability to model several problems. Some of the fundamental graph problems are graph traversals, graph connectivity, and finding a spanning tree of a given graph. In this Chapter, we study the fundamental graph problem of finding connected components of a graph on the GPU. It finds application in several other graph problems such as bi-connected components, ear decomposition, and the like. Our implementation achieves a speed-up of 9-12 over the best sequential CPU implementation and are highly scalable. Our work can thus lead to efficient implementations of other important graph algorithms on GPUs.

7.2 Related Work

There have been several PRAM algorithms for the graph connected components problem. Hirschberg et al. [54, 55] discuss a connected components algorithm that works in $O(log^2 n)$ time using $O(n^2)$ operations. However, the input representation has to be in the adjacency matrix format, which is a limitation for large sparse graphs. For sparse graphs, Shiloach and Vishkin [24] presented an algorithm that runs in $O(log\ n)$ time using $O((m+n)logn)$ operations on an arbitrary CRCW PRAM model. The input for this algorithm can be in the form of an arbitrary edge list. A similar algorithm is presented by Awerbuch and Shiloach in [10]. However, it should be noted that the PRAM model is a purely algorithmic model and ignores several factors such as the memory hierarchy, communication latency and scheduling, among others. Hence, PRAM algorithms may not immediately fit novel architectures such as the GPU.

In [48], the authors presented a wide range of optimizations of popular PRAM algorithms for connected components of a graph along with empirical results on a Connection Machine 2 and a Cray YMP/C90. Their work includes optimizations for the Shiloach-Vishkin algorithm [119] and the Awerbuch-Shiloach algorithm [10]. Though the architectures on which they reported empirical results are dated, many algorithmic observations and inferences presented are relevant to our work also. An-

other attempt at implementing connectivity algorithms was by Hsu et al [60]. Their method shows good speedups on graphs that can be partitioned well. For random graphs, no major speedups were reported.

7.3 The Shiloach Vishkin Algorithm

The Shiloach-Vishkin algorithm involves iterative grafting and pointer jumping operations. In each iteration, if (u,v) is an edge in the graph, then, under certain condition, the trees containing nodes u and v are combined to form a single tree. This process is called grafting. Further, during each iteration, pointer jumping is applied to reduce the height of the resulting trees. The algorithm terminates when all the trees in the forest are stars, and each node is assigned to one star. In each iteration the following steps are performed:

- **Grafting trees:** For each edge uv so that parents of u and v are different, one node changes parent, if parent of either u or v is the root of its tree and the parent of the other node has a lower index than the former.
- **Grafting star trees onto other trees:** This is done to reduce the depth of the resultant trees. The trees are checked to ascertain whether they are stars or not by allowing nodes which are at a depth of 2 or larger with respect to the root of the tree to mark its parent and the parent of its parents as members of non stars. Thus all the nodes which are not part of a star will be marked by this process. This step reduces the worst case complexity of the algorithm.
- **Single pointer jumping:** One step of pointer jumping is done to reduce the depth of the trees.

It is shown by Shiloach and Vishkin that this algorithm runs in $\mathcal{O}(\log n)$ time using $\mathcal{O}(m + n)$ operations. The number of iterations required for an edge to be inactive is large, hence increasing the memory bandwidth usage. This is also relevant because reading an edge list is bandwidth intensive.

7.4 A Hybrid Algorithm for Graph Connected Components

We consider finding the connected components of a given undirected graph on our hybrid computing platform. Finding the connected components of a graph is one of the fundamental graph problems with several applications. Hence, a faster, efficient hybrid multicore solution for this problem is of importance. Our strategy to solve the problem can be broadly described by the following three step process.

- Partition the graph according to a certain threshold. A higher percentage is allocated for the GPU.
- Find the connected components in each of the partitions by using both the GPU and the CPU concurrently.

- Combine the components found at the CPU and the GPU to arrive at the connected components of the input graph.

More details of our approach are given below. We partition the graph into a two parts according to a certain percentage say t. A $t\%$ nodes are processed on CPU and $(100-t)\%$ nodes are processed on GPU. These $t\%$ nodes of CPU are further split into (t/c) parts where c is the total number of cores in CPU. The CPU cores now perform sequential DFS on a subgraph corresponding to their partition. The GPU processes the subgraph corresponding to its partition using the algorithm described in Algorithm 12 and [123]. A brief pseudo-code of our approach is given in Algorithm 12. In Sections 7.5– 7.7, we describe each step of the algorithm in detail.

Algorithm 12 $ConnectedComponents(G, t)$

Input: A graph $G = (V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$ and a threshold t
Output: The number of components

1: `CPU` **::** $n_{cpu} = \frac{nt}{100}$.
2: `CPU` **::** Partition G into $c+1$ pieces $G_1, G_2, \cdots, G_{c+1}$ with $V(G_i) = \{v_{\frac{(i-1)\cdot n_{cpu}}{c}} \cdots v_{\frac{i\cdot n_{cpu}}{c}}\}$ for $1 \leq i \leq c$,
$V(G_{c+1}) = V(G) - \cup_{i=1}^{c} V(G_i)$,
$E(G_i) = E(G) \cap (V_i \times V_i)$, for $1 \leq i \leq c+1$.
3: `GPU` **::** Find the connected components of G_{c+1} on the GPU
4: `CPU` **::** Find the connected components of G_i on the ith core of the CPU, $1 \leq i \leq c$
5: Transfer GPU components to the CPU
6: `GPU` **::** Call ProcessCrossEdges

To partition the graph we just consider the first $t\%$ of the nodes in one partition and the remaining nodes in the other partition. There may be better ways to partition the graph so as to minimize the overall time, but those methods may be more time consuming in general. The approach we follow can be seen as a case of MIMD data parallelism where different functions are applied on different data sets. This is necessitated by the fact that the best algorithm suitable for a CPU may be different from the best algorithm that is suitable for a GPU. For instance, solutions such as DFS/BFS are not very efficient on architectures such as the GPU. Therefore we use the Shiloach-Vishkin (SV) algorithm [119] on the GPU. Similarly, the overhead of the SV algorithm compared to a DFS/BFS kind of solution makes it unsuitable on the CPU.

When finding the connected components of subgraph G_i, for $1 \leq i \leq c+1$, we only consider edges e such that both the endpoints of e are in G_i. We call edges where the endpoints lie in different subgraphs as *cross edges*. Figure 7.1 shows an example of cross edges. These cross edges are considered in Algorithm 13 described in Section 7.7.

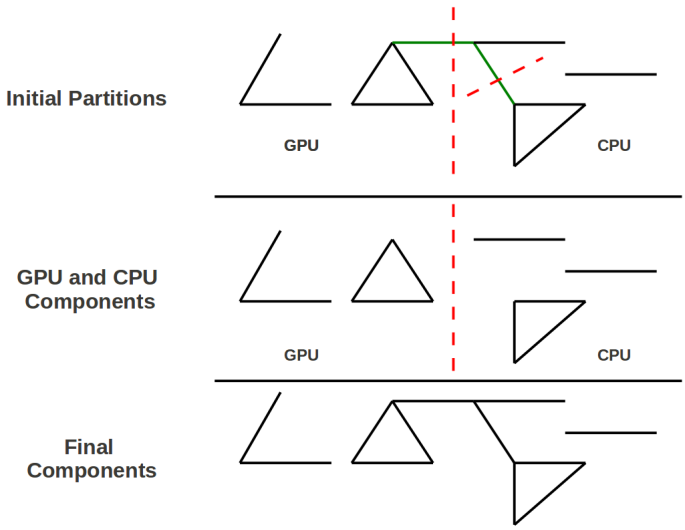

Figure 7.1 An example run of the algorithm. The red dashed lines denotes the partitions within the GPU or the CPU cores. The green edges are the cross-edges.

7.4.1 Example

An example run on a small sample graph has been shown in Figure 7.1. As we can see,that the red dashed lines denotes the static partitions that are initially created amongst the two devices and also among the various cores of the CPU (two in this case). In the next step, we observe the computation of the components on each of the devices. The cross edges are denoted by green and are recorded in the first phase during partitioning. In the final phase, these cross edges are used to fuse the components together to form the final result.

7.5 The Shiloach Vishkin Algorithm for GPU

For finding the connected components of G_{c+1} on the GPU, we utilize a modified version of the well known Shiloach- Vishkin(SV) algorithm [119].

Typically, connected components algorithm is an irregular memory access algorithm which is actually considered to be highly unfit for a GPU implementation as stated in [123]. In order to reduce the irregularity of the operations of the GPU computation the following optimizations in the Shiloach-Vishkin algorithm are done.

- Removing atomic operations
- Reducing 64 bit read overhead by packing 32 bit reads.
- Allowing partial results from previous iterations to be carried forward

The implementation from [123] on the GPU follows three main steps: 1) Hooking 2) Pointer Jumping 3) Graph Contraction. These three steps are briefed below.

- **Hooking:** The selection of the parent nodes for the hooking process is randomized and the sensitivity towards vertex labels is reduced. This reduces the overhead of the whole process. Also, in the even iterations the node with the lower label selects the node with the higher ones as its parent and the reverse happens in the odd iterations.
- **Pointer Jumping:** In the original Shiloach-Vishkin Algorithm single step pointer jumping was proposed. However, we apply complete pointer jumping in order to convert the tree into a rooted star in a single iteration. Now, the nodes of the tree are only present at two levels: the root level and the leaf level.
- **Graph Contraction:** This process is achieved through edge hiding. As the edges are active only in the hooking step, the edges are not activated for any further processing once hooking is complete. In this way, the data movement is reduced thereby reducing the overheads involved.

The above GPU-specific optimizations to the Shiloach-Vishkin algorithm were introduced in [123] and presently the results of [123] outperform the other GPU-based connected-component algorithms.

7.6 DFS on CPU

For finding the connected components of the graph G_i for $1 \leq i \leq c$ on the ith core of the CPU, we use the standard DFS algorithm [28]. This is motivated by the fact that since the available parallelism on the CPU is small, highly data parallel algorithms do not make a good fit. Further, each CPU core can run independently minimizing any overheads in synchronization and communication. The output of this step is that each CPU core labels the components identified uniquely.

7.7 Processing Cross Edges

After finding the connected components of each G_i for $1 \leq i \leq c+1$, we construct a supergraph as follows. Each of the components identified so far is represented by a super-vertex which is typically the lowest numbered vertex in that particular component. Each of these super-vertices may be connected by the cross-edges which we had earlier identified. Of the entire set of cross edges that connect a pair of supervertices, we select only one such cross edge. We now run a parallel connected components algorithm on this super-graph to identify the components which are connected by the cross-edges to give us the final result.

We now present Algorithm 13 that identifies connected components in the supergraph. In Algorithm 13, we run threads for each of the cross edges of the supergraph S. Each of the threads checks the labels of the end-points of the cross-edge. Here, label refers to the mark on each of the nodes which indicates

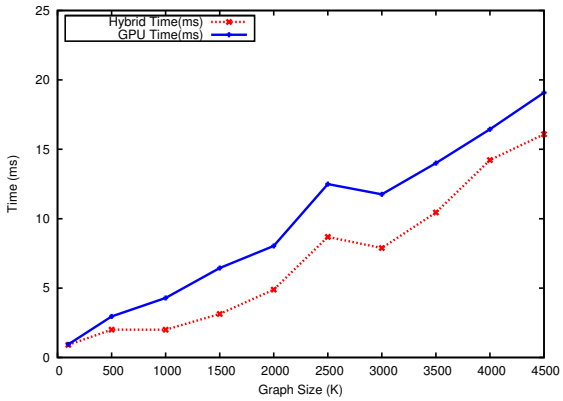

Figure 7.2 Time comparison with graphs sizes varying from 500K to 4.5M nodes. The GPU times are obtained by running the code from [123] on the same instances.

the super-vertex it belongs to. If the labels of the end-points differ, the labels of the two supervertices, and hence two components, are set to the minimum of the two labels. This unifies the two components. This is repeated until no labels can be updated.

Algorithm 13 $ProcessCrossEdges(Supergraph\ S)$

1: **while** Labels change **do**
2: **for** each of the cross-edges uv where $uv \in S$ **do**
3: `GPU` `::` $label(u) = min(label(u), label(v))$
4: `GPU` `::` $label(v) = min(label(u), label(v))$
5: **end for**
6: **end while**

7.8 Results

Our experimental set up is as described in Chapter 3. In this work, we consider random graphs of several components. These random graphs are generated according to the $\mathcal{G}(n, p)$ model [22] using the GTgraph generator [13].

In Figure 7.2 we show the speed-up achieved by our hybrid algorithm on several graph sizes. The label "Hybrid Time" refers to the runtime of Algorithm 12. The label "GPU Time" refers to the time obtained by the code from the work of [123] on the same GPU as is used in our computing platform. Both these implementations are run on the same datasets generated as described in the previous paragraph. As can be seen from Figure 7.2, the hybrid algorithm consistently outperforms the best known GPU implementation by about 25% on average. In all the results presented in Figure 7.2, and also elsewhere in this section, the experiment is run for multiple runs and the average result is considered. In Figure 7.2, the runtime of the hybrid algorithm reported are the figures which corresponds to the best thresholds.

In our next experiment, we study the relationship between the threshold and the speed-up achieved on a graph of a particular size. In Figure 7.3 we show the speedup obtained for a 1.5 million, and a 3 million node graph at various thresholds. As expected we see from Figure 7.3 that the speedup is initially negative and then increases. From Figure 7.3 we see that the speedup reaches a maximum at around the 25% threshold for 1.5 million nodes and 17% for 3 million node graph.

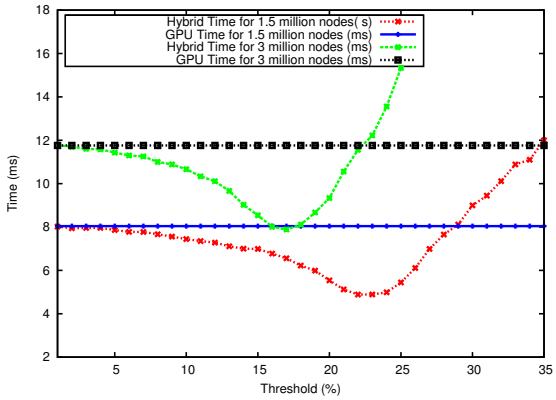

Figure 7.3 Time comparison in graphs of size 1.5 million and 3 million nodes. The GPU timings are the obtained by running the code from [123] on the same instances.

We also studied the effect of the graph size on the threshold. For this purpose, we vary the size of the graph and find the threshold that gives best performance. The results of this study are shown in Figure 7.4. It can be noticed that the threshold reduces as the size of the graph increases. This can be explained by the fact that the load on the CPU is comparable on small graphs when the threshold is high. As the size of the graph goes up, this same load is reached on smaller thresholds. This exact behavior is reflected in Figure 7.4 where the threshold varies accordingly.

It can be noted that the Algorithm 12 when run with a threshold of 0% will correspond to the algorithm of [123]. This can be seen as the *pure GPU* algorithm. For the connected components problem, the improvement in the performance is therefore due to the hybrid computing platform. To illustrate this further, in Table 7.1 we show the time taken by the CPU and the GPU in our hybrid algorithm for a graphs of varying sizes. The runtimes correspond to the optimum threshold of partitioning the graph. This optimum threshold is identified emprically by iterating over a range of thresholds. It can be seen from Table 7.1 that at the optimum threshold, the time taken by the GPU and the CPU are close to each other. Additionally, the GPU is never idle except in two instances, and the CPU has no more than 5% idle time.

7.9 Static Auto-tuning

Our experimental observations in the previous section imply that the size of the graph has some bearing on the right threshold to use for best performance. As can be seen from Figure 7.3 , for a graph

Table 7.1 Timing for finding connected components and cross-edge processing. All timings are in ms and graph sizes in M

Graph Size	GPU Time	CPU Time	Processing Cross-Edges	Total
1	4.13	4.03	0.01	4.14
1.5	4.30	4.23	0.01	4.31
2	7.64	7.71	0.02	7.73
2.5	13.80	13.65	0.02	13.82
3	12.89	12.76	0.03	12.92
3.5	14.73	14.55	0.03	14.76
4	17.61	17.70	0.05	17.75
4.5	19.91	19.34	0.06	19.97

of 1.5 M vertices, a threshold of 25% is good whereas for a graph of 3 M vertices, the threshold reduces to 17%. In general, one can ask what threshold to choose for a given graph. To this end, we performed our experiments on a range of graph sizes and a range of thresholds to identify the threshold that gives the best performance for a given graph size. This best threshold with respect to various graph sizes is shown in Figure 7.4. One can use the information from Figure 7.4 to know the right threshold to use for any given graph size by using standard interpolation techniques if required. This resembles the static auto-tuning model where certain parameters can be chosen according to specific input characteristics before launching the parallel program. A similar kind of an experiment is performed where the number of vertices in fixed at 2.5 M and the number of edges is varied. The result of this experiment is shown in Figure 7.5.

One aspect to note from Figure 7.4 and Figure 7.5 is that the threshold decreases with increasing graph size. This is due to the fact that the rate at which the CPU in our computing platform can process edges is slower than the rate at which the GPU can. Hence, as the graph size increases, the portion of edges that the CPU has to process decreases.

7.10 Conclusions and Future Work

In our present work, we considered a very simple hybrid computing platform with only one multicore CPU and a GPU. Our hybrid algorithms however are designed to scale when more CPUs or more GPUs are added to the computing platform by making appropriate changes. These changes are necessary in the steps of static partitioning of the graph as well as the consolidation step where the components computed by each CPU core or GPU is merged together based on the distributed cross-edges. In this scenario, usually the communication bottleneck between the multiple nodes can prove to be an overhead. So, keeping local copies of the cross edges in each of the GPU or CPU before the start of the computation can help reduce this overhead.

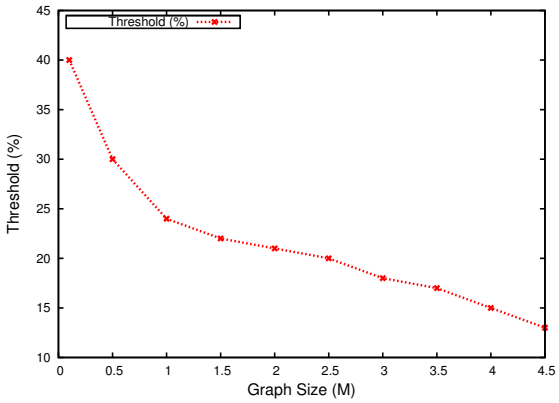

Figure 7.4 Threshold variation on graphs with sizes varying from 500K to 4.5M

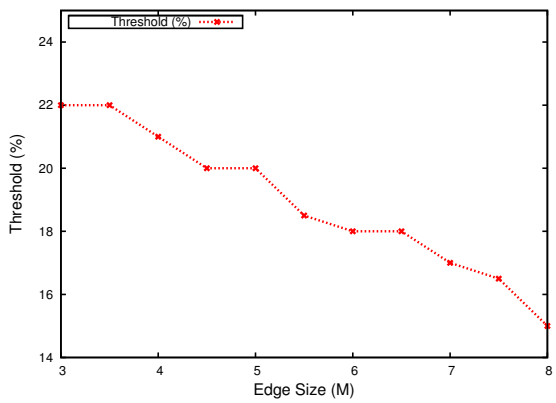

Figure 7.5 Threshold variation on a graph of 2.5 million nodes and edge sizes varying from 3M to 8M

As hybrid algorithms become popular, it is also important to consider new programming mechanisms that allow for implementing hybrid algorithms efficiently. For instance, mechanisms to improve the synchronization support across devices can help programmability of hybrid algorithms.

In future, we wish to consider further fundamental problems for which one can design hybrid algorithms. Other analytical issues that can be considered are: a mechanism to arrive at an optimal assignment of tasks to processors, a notion of critical path in such an assignment, and the like.

Chapter 8

Parallel Breadth First Search

8.1 Introduction

Graph algorithms find a large number of applications in engineering and scientific domains. Prominent examples include solving problems arising in VLSI layouts, phylogeny reconstructions , data mining, image processing, and the like. Some of the most commonly used graph algorithms are graph exploration algorithms such as Breadth First Search (BFS), and computing components. As the current real life problems often involve the analysis of massive graphs, it is often seen that parallel solutions provide an acceptable recourse.

Heterogeneous algorithms that aim to utilize all the computational devices in a commodity heterogeneous platform have also been designed for graph breadth-first exploration [58, 91, 43]. Most of these use platforms consisting of multicore CPUs and GPUs. All of the above-cited works show an average of 2x improvement over pure GPU algorithms.

Most of the above works in general aim at data structure optimizations but largely run classical algorithms on the entire input graph. These algorithms are designed for general graphs whereas the current generation graphs possess markedly distinguishable features such as being large, sparse, and large deviation in the vertex degrees. In Figure 8.1, we show some of the real-world graphs taken from [2]. As can be seen from Figure 8.1, these graphs have several vertices of very low degree, often as low as 1. For instance, in the case of the graph web-Google, 14% of the vertices have degree 1. Table 8.1 lists other properties of a few real world graphs from [2].

Current parallel algorithms and their implementations [91, 43, 112, 58, 135] do not take advantage of the above properties. For instance, in a typical implementation of the breadth-first search algorithm, one uses a queue to store the vertices that have to be explored next. But, a vertex v of degree 1 that is in the queue will not lead to the discovery of any yet undiscovered vertices. So, the actions of BFS with respect to v such as adding it to the queue, dequeue it, and then realize that there are no new vertices that can be discovered through vertex v are all unnecessary. These actions unfortunately can be quite expensive on most modern parallel architectures as one has to take into account the fact that the queue is

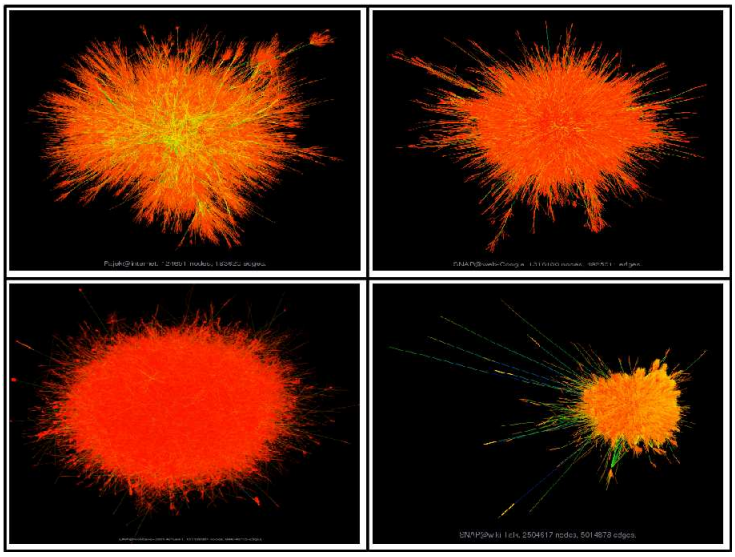

Figure 8.1 A sample of four real world graphs from [2]. On the top-left corner is the graph internet, top-right is the graph web-google, bottom left is the graph webbase_1M, and the bottom-right is the graph wiki-Talk.

to be accessed concurrently. Similarly, other operations such as checking of the status of a vertex, may be quite disposable.

In light of the above paragraph, we envisage that new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multicore architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. In this Chapter, we propose graph pruning as a technique in this direction. Graph pruning aims to reduce the size of the graph by pruning away certain elements of the graph. The required computation is then performed on the remaining graph. The result of this computation is then extended to the pruned elements, if necessary.

In this Chapter, we apply the graph pruning technique to two important graph algorithms: Breadth-first search, and connected components. In each case, we show that pruning pendant nodes iteratively can result in reducing the size of the graph on real-world datasets, by as much as 25% in some cases. This reduction in size helps us achieve remarkable improvements in speed for the above two workloads by an average of 35%.

8.1.1 Related Work

Many recent works in parallel computing have focused on graph algorithms. Few among them include [25, 58, 139, 91]. The work of Scarppaza et al. [112] demonstrates the use of an all-to-all exchange of visited nodes information across the eight SPUs of a Cell BE. One of the first results of BFS using GPUs is the work of Harish et al. [49]. Subsequent improvements to [49] centered around the use of heterogeneous computing. In [58], Olukotun et al. use a CPU+GPU platform where the

levels of the BFS with fewer discovered nodes are processed on the CPU and levels with large number of discovered nodes are processed on the GPU. Using such a heterogeneous strategy, they achieve a throughput of 0.4 Beps (Billion edges per second) on Erdos-Renyi random graphs. These are improved further by Bader et al. [91]. Some of the prominent works on multicore CPUs include [25] where the primary goal is to map the data structures to the cache hierarchy so as to improve the cache hit rates. A recent work [43] partitions the graph so that low degree vertices are processed on the GPU and the high degree vertices are processed on the CPU.

Graph connectivity on symmetric multiprocessors (SMP) is studied by Bader and Cong in [12]. Their idea is to give each processor a stub spanning tree to which unvisited nodes may be added iteratively. In a recent work, Harish and Narayanan [49] have implemented a modified BFS style graph traversal on the GPU. Their implementation works with a sorted edge list as the input. There have also been other efforts to implement parallel BFS on different architectures (cf. [112, 143, 47]).

Finding the connected components of a graph also is an important primitive and hence has attracted a lot of attention within the parallel computing community. Popular parallel algorithms in the PRAM model include the algorithm of Shiloach and Vishkin [119] and its variants by Greiner [48]. On GPUs, a variant of Shiloach and Vishkin [119] is used by Soman et al. [123]. A heterogeneous execution of this algorithm on a CPU+GPU platform with an improvement of 35% on average is shown in [16].

Algorithm or implementation decisions based on the nature of the graph is an emerging area of research. In [37],the authors propose a Distributed Leaf Pruning (DLP) strategy that helps in achieving a significant speedup over distributed communication networks. In this work, the authors noticed that in many real life networks, like CAIDA, the average node degree of a graph with n nodes is very less than n and nodes with an unitary degree is typically high. So, pruning these nodes from the graph, provided a much better performance in packet forwarding strategies over the entire network.

In [103], the authors show novel pruning techniques that solves the maximum clique problem on large sparse graphs. Their main idea is to prune the vertices that strictly have fewer neighbors than the size of the maximum clique already computed. These are the vertices that can be exempted from the computation as, even if a new clique is found, its size would not be greater than the maximum one that is already computed.

Input pruning has been used as a technique in the design of *work-optimal* parallel algorithms in the PRAM model. Popular examples include the list ranking algorithm of Anderson and Miller [8], the optimal merging algorithm [26], the optimal range minima algorithm [114], and so on. In all of these cases, the size of the input is reduced to an extent after which a slightly non-optimal algorithm is employed. In a post-processing phase, the results on the reduced input is extended to obtain a result for the entire input.

8.1.2 Our Results

In this work, we focus on graph BFS, and connected components. For these two graph algorithms, we first show that a similar preprocessing phase can help reduce the size of the graph by an average

of 35% on a wide variety of real-world graphs. This helps us to obtain an average of 40% speed-up compared to the best known implementations for the above problems on similar platforms.

Our preprocessing simply involves removing pendant nodes from the graph. This is done iteratively so that nodes on pendant paths are also removed during preprocessing. In the post-processing phase, we show that extending the output of the computation on the smaller graph can be done in a very straightforward and quick manner.

Some of our specific contributions are as follows:

- Our results improve the state-of-the-art for graph BFS by 35%. We achieve an average throughput of 2 billion edges per second on a wide range of data sets including graphs from the University of Florida collection [2], and graphs generated using the Recursive Matrix Model (R-MAT). The R-MAT generator is efficiently implemented in the GTGraph suite[13].

- On the connected components problem, we get an average 20% improvement over the best known result on an identical platform [16]. A small change to the algorithm can also build a spanning tree of a graph with very little extra time.

8.2 Our Approach

In this section, we present a three phase technique, outilned in Algorithm 14, for scalable parallel graph algorithms of real world graphs. In the first phase, called the *preprocessing phase*, we reduce the size of the input graph by removing *redundant* elements of the graph. Once the graph size reduces, the second phase involves using existing algorithms to perform the computation on the smaller graph. In a final phase, we then extend the result of the computation to the entire original graph via quick post-processing, if required.

Let G be a large, sparse graph. As mentioned in Algorithm 14, let Prune(G) be a function that can prune certain elements of G. Let G' be the graph that remains after Prune(G). Let A be an algorithm that can compute the desired solution. We then use algorithm A on the graph G'. Let O' be the solution on G'. In a post-processing third phase, we extend the solution O' to a solution O of the entire graph G.

Algorithm 14 $ProcessGraph(Graph\ (\mathcal{V}, \mathcal{E}))$

1: **/* Phase I – Prune */**
2: $G' =$ Prune(G)
3: **/* Phase II – Compute */**
4: $O' = A(G')$
5: **/* Phase III – Extend */**
6: $O =$ Extend(G, O')

We note that if Phase I prunes only a constant fraction of the size of the graph, and one uses a standard algorithm in Phase II, then the asymptotic runtime using the above technique is still unchanged. How-

ever, even such a constant fraction reduction in size can have a considerable impact on the experimental efficacy.

We envisage that different graph algorithms can benefit from corresponding pruning processes in Phase I. Further, step 1 may also be performed iteratively. Each iteration may prune some nodes after which more nodes may become candidates for pruning in the next iteration. We refer the reader to Algorithm 15 for an illustration. In Algorithm 15, $\mathcal{P}$ refers to a property that vertices that are pruned will satisfy. Similarly, the post-processing in Phase III can also be based on the problem at hand. If Phase I is spread over multiple iterations, then Phase III may also be spread over multiple iterations, possibly in the reverse order of iterations of Phase I.

Algorithm 15 $Prune(Graph\ (\mathcal{V}, \mathcal{E}))$

```
1: for i = 1 to r iterations do
2:     for each vertex v ∈ G do
3:         if v has property P then
4:             Remove v, and all edges incident on v.
5:             Store (v, i) for future re-insertion step.
6:         endif
7:     endfor
8: endfor
```

It is important to note that the property $\mathcal{P}$ can be evaluated quickly. This helps keep the overall time for Phase I small. The time taken by a graph algorithm using our technique will depend on the extent of pruning achieved in Phase I and also the time taken in Phase I and III. As can be noticed, in most cases, there will also be a trade-off between time taken by Phase I and III and that of Phase II. In fact, such a trade-off is observed in the case of list ranking [16].

Some of the properties that may be of interest are the following.

- Pendant nodes: Let us call a node v in a graph G as a pendant node if the degree of v is G is 1. For the two workloads we consider in this Chapter, we show that a simple pruning based on removal of pendant nodes suffices. This is also the pruning technique used in [37].
- Independent nodes: A subset of nodes is called as an independent set of nodes if they are mutual non-neighbors. This has been used in list ranking algorithms [8] and its recent heterogeneous implementation [139, 16].
- Graph partitioning: Graph partitioning calls for partitioning a graph G into a specified number k equal partitions such that the number of edges that have end points in different partitions is minimized. In an influential work, Karypis and Kumar [68], introduce the coarsening-refinement approach. During the coarsening step, a matching of the current graph is computed and prunes matched vertices.

The above examples indicate that various properties $\mathcal{P}$ can have applicability to different problems. Thus, our approach is quite general. It must be noted that all the above examples are not implemented

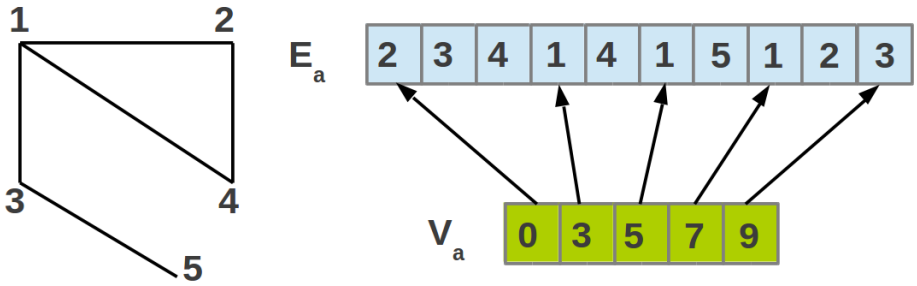

Figure 8.2 The CSR format for representation.

on modern parallel architectures. In this Chapter, we show that our technique can be used on modern parallel architectures too.

8.3 Breadth First Search

Breadth First Search (BFS), is one of the most widely used graph algorithms and finds massive applications in the domains of state space partitioning, graph partitioning, theorem proving, and networks. The problem statement of the BFS is: given an undirected, unweighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and a source vertex $\mathcal{S}$, compute the minimum number of edges that are needed to reach every vertex of $\mathcal{G}$ from $\mathcal{S}$. The optimal sequential solution to this problem runs in $\mathcal{O}(\mathcal{V} + \mathcal{E})$ time [28].

The well known sequential algorithm maintains a queue where the newly discovered vertices are are inserted at the rear. Current vertices are deleted from the front of the queue and this process continues until the queue is depleted. All the newly visited vertices are constantly enqueued along the way. For representation of the graph in the memory, we use the compact adjacency list which is more popularly known as the compact sparse representation (CSR). An example is shown in Figure 8.2. In CSR, all the adjacency lists are packed into a single large array. An array E_a is used to store the adjacency lists where the list for vertex $i + 1$ immediately follows vertex i, for all the vertices in $\mathcal{G}$. An array V_a, stores the starting indices of the corresponding adjacency lists in E_a. Each of the indices of V_a acts as the vertex number of of the graph. The key advantage of using this representation is that, the graph is stored in a contiguous memory locations and no long strides are required to go from a neighbor of a certain vertex. This helps in reducing the memory access irregularity and hence boosts the overall performance of the BFS implementation.

8.3.1 Implementation

The basic approach of our algorithm is to first perform a pruning step where pendant vertices are removed iteratively. This is followed by an efficient parallel execution of BFS on the CPU+GPU hybrid

platform from Chapter 3. Finally, in a post-processing step, the level number for vertices that were removed initially will be is computed. Our algorithm is described in Algorithm 16.

Algorithm 16 $BFS(Graph\ (\mathcal{V}, \mathcal{E})), VertexS$

1: Call Algorithm 15
2: Perform BFS in hybrid on GPU and CPU (See Algorithm 17)
3: **for** $i = r\ to\ 1$ **do**
4: Re-insert removed nodes according to (i, v) information previously stored.

8.3.2 Phase I

The first phase of removing the pendant vertices is done entirely on the GPU as it is a purely parallel step with no irregular memory operations involved. Hence, a hybrid implementation of this step puts an unnecessary overhead of data transfer. We however add the following optimizations.

- To reduce the time spent in Phase I, we use the CSR representation and identify pendant vertices as follows. Consider vertices u and v numbered consecutively. Then, u is a pendant vertex if $V_a[u]$ and $V_a[v]$ differ by 1. If vertex u is numbered n, then the above rule has to be modified to say that $V_a[n] = |E_a|$. See Figure 8.2 for an illustration where vertex 5 is a pendant vertex, and also $V_a[5] = 10$. In essence, threads need not read the E_a array, and also do not have uncoalesced accesses.

- Notice that if a node v that is removed in iteration i, then its only neighbor can now become a pendant vertex in iteration $i+1$. Further, in iteration $i+1$, we need to check only such vertices w. Therefore, we mark such w in iteration i, and do not check other vertices in iteration $i + 1$. This helps in reducing the time spent in Phase I across each iteration. For illustration, see Figure 8.2. If vertex 5 is removed in the first iteration, then vertex 3 is marked as a potential pendant vertex in that iteration. Since the remaining degree of vertex 3 is now 1, also vertex 3 can be removed in the second iteration.

8.3.3 Phase II

We now present our detailed algorithm in Algorithm 18 that is used in Phase II. Algorithm 17 is similar in spirit to the one used by Munguia et al. [91]. The label `CPU::` and `GPU::` in Algorithm 17 refer to steps executed on the CPU and the GPU respectively. The CPU and the GPU maintain array VISITED, FR, and NFR which contain the visited vertices, the current frontier vertices, and the next frontier vertices, respectively. The array VISITED is shared between the two devices so that the status

Algorithm 17 $PhaseII(Graph\ (\mathcal{V},\mathcal{E}), Vertex\ S)$

Input: A graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and starting vertex $\mathcal{S}$
Output: Level numbers of all the vertices.

Set a threshold for separating the vertex set between CPU and GPU.
Create G_c graph for CPU and G_g for GPU.
Create initial FR array with S
while FR ! $= \phi$
    `GPU` **::** Call GPU_BFS$((\mathcal{G}',\mathcal{S}), Array\ FR)$ (Algorithm 18)
    `CPU` **::** Perform CPU BFS [28].
    Check NFR array of GPU and CPU for termination
    Set FR := NFR
endwhile
Consolidate LEVEL values

Algorithm 18 $GPU\_BFS\ (Graph\ (\mathcal{V},\mathcal{E}),\ Vertex\ S,\ FR)$

Input: A graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and starting vertex $\mathcal{S}$
Output: Level numbers of all the vertices.

tid=threadID
Initialize LEVEL[S]=1;
Set NFR to NULL
Calculate $range$ in FR based on size of E_g and threads
Find $start$ and $stop$ of index of vertices from $range$
for $i\ = start\ to\ stop$
    VISITED[i]=1
    for $j \in$ all neighbors of i **do**
        if(VISITED[j] == FALSE)
            LEVEL[j]=LEVEL[j]+1;
            Add j to NFR;
        endif
    endfor
endfor

UF Sparse Matrix Collection				
Graph	**Nodes**	**Edges**	**Pendant Vertices**	**r**
internet	124,651	207,214	56,959 **(45%)**	2
dblp_2010	326,183	807,700	87,881 **(26%)**	3
watson_1	386,992	1,055,093	44,637 **(11%)**	2
webbase_1M	1,000,005	3,105,536	80,053 **(8%)**	4
wiki-Talk	2,394,385	5,021,410	176,617 **(7.37%)**	4
web-Google	916,428	5,105,309	134,452 **(14.6%)**	3
rail2586	923,269	8,011,362	117,342 **(12.7%)**	4
tp-6	1,014,301	11,537,419	194,764 **(19%)**	4
R-MAT Graphs				
rmat_1	131,072	256,784	43,556 **(33.2%)**	1
rmat_2	263,144	554,678	67,345 **(25.6%)**	2
rmat_3	525,288	1,048,515	55,453 **(10.5%)**	2
rmat_4	1,056,534	2,097,345	78,234 **(7.4%)**	3
rmat_5	2,045,266	4,190,223	92,345 **(4.5%)**	4
rmat_6	3,074,344	8,456,240	105,117 **(3.4%)**	3
rmat_7	3,156,834	12,673,552	132,443 **(4.2%)**	4
rmat_8	3,765,223	16,673,993	153,442 **(4.1%)**	4

Table 8.1 The graphs used for experimentations and their properties. The column heading r in the last column indicates the number of iterations required to remove all pendant vertices.

of a vertex can be polled whenever required. This sharing is made possible via the use of host side allocation through pinned memory. The array LEVEL is used to store the number of edges in a shortest path from s to every other vertex in the graph. This arrays is maintained locally at both the CPU and the GPU and is merged at the end of each iteration. An iteration corresponds to exploring all the vertices in the current frontier, given by the FR array. Once either the CPU or the GPU checks that the NFR array that it is maintaining is already explored by the other device. The execution stops and the LEVEL array is transferred from the GPU to the CPU.

The entire algorithm executes until the current FR array is not empty, that is all the vertices has been visited. To this end, both the CPU and the GPU work in a synchronous manner to perform the exploration. The GPU does a thread based partitioning of the adjacency list E_a. The CPU on the other hand does a more coarse grained execution on the E_c portion using the same algorithm using all the threads available to it with simultaneous multithreading. The GPU BFS part maintains a frontier array FR, which is the queue where it continually deletes elements from and also maintains a NFR which is the next frontier to be visited. Both the CPU and the GPU maintains this NFR information locally so as to minimize the communication overheads. To further minimize the cost of communication, we transfer the NFR in an asynchronous manner so that maximum amount of overlap can be achieved with the communication and the devices stay idle for the minimum amount of time. After each iteration, both the CPU and the GPU communicates this NFR array and sets it as its current FR if it has not been already visited by the other device. When the BFS algorithm on both CPU and the GPU terminates, the two devices consolidates their LEVEL arrays.

8.3.4 Phase III

In Phase III of the algorithm, we re-insert the vertices that were removed in Phase I as follows. Let v be a vertex removed in iteration i of Phase I, and let w be the only neighbor of v prior to its removal. Then, the LEVEL number of v is set to be one more than the LEVEL number of w. Further, nodes removed in iteration i are processed before those removed in iteration $i-1$. This entire step like Phase I is done entirely on the GPU because of the higher degree of parallelism that enables us to run one thread per node.

An example run of our algorithm is presented in Figure 8.3 for the graph from Figure 8.2.

8.3.5 Results

In this section, we present the results of our implementation. We compare the results with those of [91]. The results of [91] are the currently reported best results for graph breadth first search on identical platforms. In all of the results we do not time the initial transfer of the transfer of the GPU edges.

In Figure 8.4, we run our implementation on a sample of eight real world graphs from the University of Florida dataset [2]. As shown in Figure 8.4, our results outperform the results of [91]. Similar effect is seen also on random graphs generated using the R-MAT generator [13] as shown in Figure 8.5. These

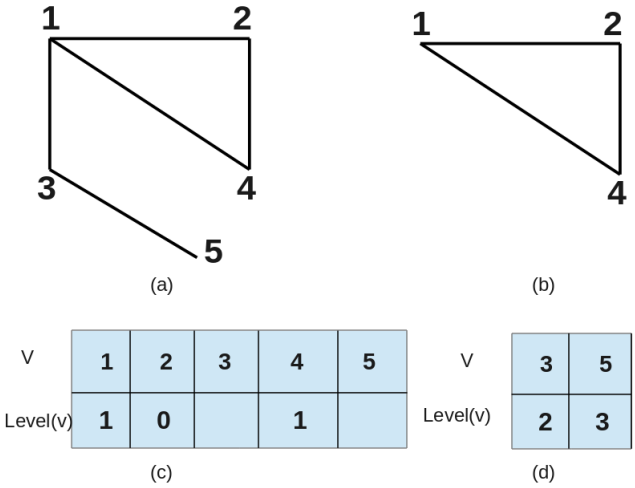

Figure 8.3 An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

graphs are generated with the default values for $a = 0.45$, $b = 0.15$, $c = 0.15$, and $d = 0.25$ of the R-MAT as set by the authors as they represent many real world graphs. All the experiments were carried out on the same platform using the code that was obtained from the respective authors of the papers. Also, the results reported are averaged over multiple executions.

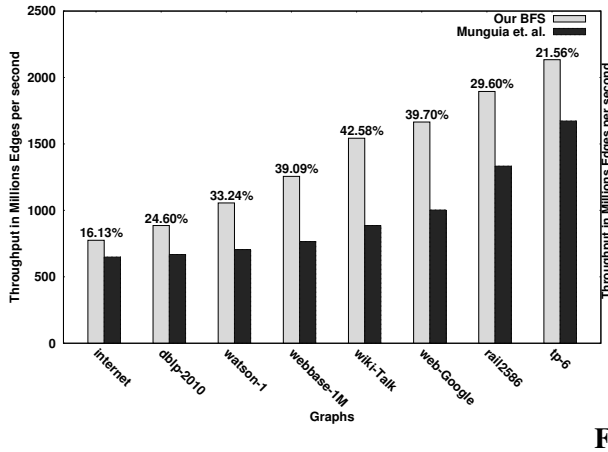

Figure 8.4 Performance of BFS on the UF Sparse Matrix dataset. The numbers show the percentage improvement.

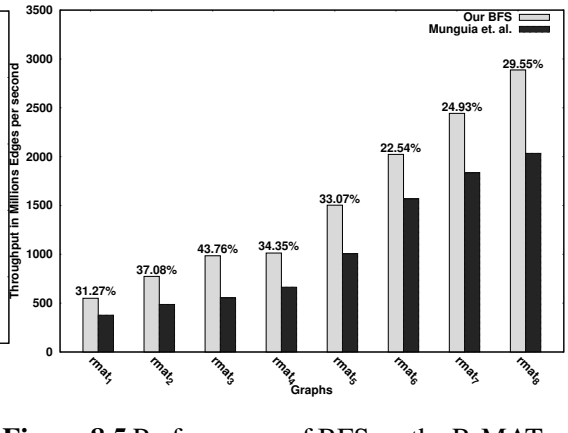

Figure 8.5 Performance of BFS on the R-MAT dataset. The numbers show the percentage improvement.

To analyze the results we obtain we perform two further experiments. We study the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 8.6 for the graphs webbase_1M, wiki-Talk and web-

Google from the dataset of [2]. Figure 8.6 percentage of vertices are pruned from the input graph. The improvement can be attributed to the lesser number of operations required in our implementation as pruned vertices do not enter/exit arrays FR, VISITED, and NFR.

We also experimented on the trade-off between the number of iterations of Phase I and the overall runtime. As an example, consider that the graph webbase_1M from the [2] dataset. Notice from Figure 8.7 that in the sixth iteration of Phase I, only 134 pendant vertices can be removed. The question we seek to answer is whether one should run one more iteration of Phase I to remove these 134 vertices, or move on to Phase II.

Figure 8.7 shows the results of the above experiment on the webbase_1M graph from the [2] dataset. It can be noticed that the time taken for each iteration of Phase I does decrease. This is attributed to the fact that successive iterations do not test each vertex whether it can be removed. Rather, we flag potential pendant vertices in an iteration so that only those vertices are checked in the next iteration. The number of nodes that are removed in each of the iterations are plotted over the Phase I curve. The overall runtime decreases initially as we expect to remove more pendant vertices in the first few iterations of Phase I. The time for Phase I is shown on the right-side Y-axis of Figure 8.7. The important observation we can note from Figure 8.7 is that the overall runtime plateaus after five iterations of Phase I for the graph considered. Therefore, it is worthwhile to stop Phase I once the number of remaining pendant vertices reduces below a small threshold of say 500.

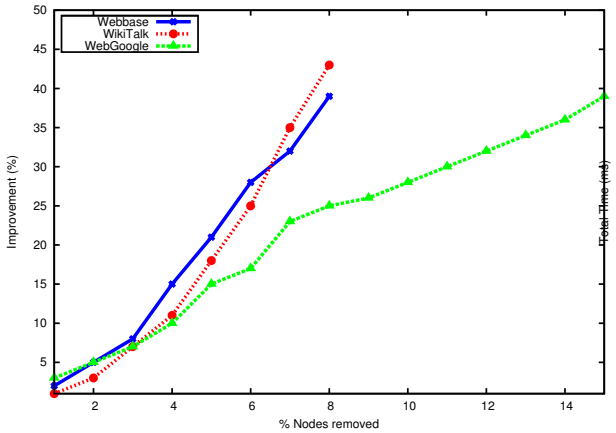

Figure 8.6 Percentage improvement of performance of BFS as a function of the percentage of pendant nodes removed in Phase I.

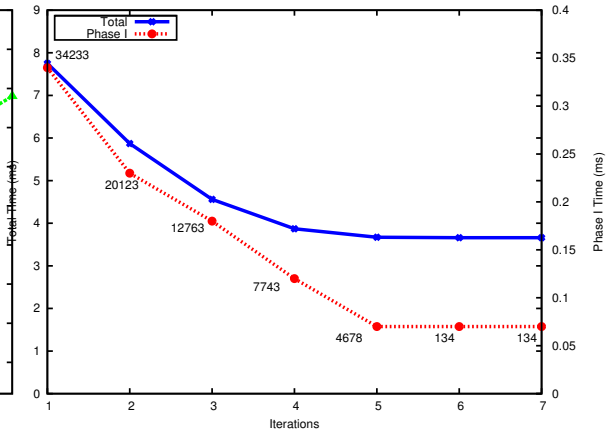

Figure 8.7 Trade-off between Phase I and the overall runtime of BFS.

8.4 Connected Components

Finding the connected components of a graph is of fundamental importance to graph algorithms. Given a graph $G = (V, E)$, the problem is to find a partitioning of V into disjoint sets $V_1, V_2, \cdots$, so that vertices u and v are in the same set if and only if there is a path between u and v in G. Well known sequential algorithms such as the Depth First Search algorithm (DFS) [28] run in $O(n + m)$ time. Several efficient parallel algorithms in the PRAM model have been proposed. Popular among them are the algorithms of Shiloach and Vishkin [119], and the algorithm of Greiner et al. [48]. However, because of the irregular nature of operations involved, this workload is often difficult to implement on most modern parallel architectures. Efficient implementations of the Shiloach and Vishkin algorithm are known to exist for a variety of parallel architectures including symmetric multiprocessors [12], Cray and CM2 [48], GPUs [123], and also on CPU+GPU systems [16].

In this section, we apply techniques from Section 8.2 and show that highly efficient hybrid algorithms can be designed for this workload on the CPU+GPU hybrid platform described in Chapter 3. Our solution can be broadly outlined in the following steps and follows the algorithm used in [16].

8.4.1 Implementation

The main steps of our implementation is outlined in Algorithm 19.

Algorithm 19 $Connected\_Components(Graph\ (\mathcal{V}, \mathcal{E}))$

Input: A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Here $|V| = n$.
Output: Labels of each vertex identifying its component.

1: Call Algorithm 15
2: Initialize LEVEL[S]=1;
3: Set a threshold t.
4: `CPU` **::** $n_{cpu} = \frac{nt}{100}$
5: `CPU` **::** Partition E into E_{cpu} and E_{gpu} where E_{cpu} is the edges corresponding to n_{cpu} nodes and E_{gpu} is the rest.
6: Find connected components of the graph $G(V - n_{cpu}, E_{gpu})$ on the GPU using the Shiloach-Vishkin algorithm [119], and the connected components of $G(n_{cpu}, E_{cpu})$ on the CPU using DFS. The graph $G[n_{cpu}]$ is further divided into c equal partitions where c is the number of threads run on the CPU.
7: Use the cross-edges recorded during the partition phase to compute the final components.
8: Re-insert the edges removed in Step 1.

In Step 1 of Algorithm 19, we *prune* the pendant nodes in the given graph iteratively. This is done by the sequence of steps as outlined in Algorithm 15.

In the next steps, we partition the graph according to a predetermined threshold t. The optimal value of t is later on determined experimentally. We divide the edges of the graph so that the first $t\%$ is in one partition and the rest in the other. We allocate the smaller partition to the CPU and the other to the GPU. The partitioning strategy that we follow is can be defined as a case of MIMD data parallelism where different functions are applied to the different data sets. This is because the algorithm that is well suited

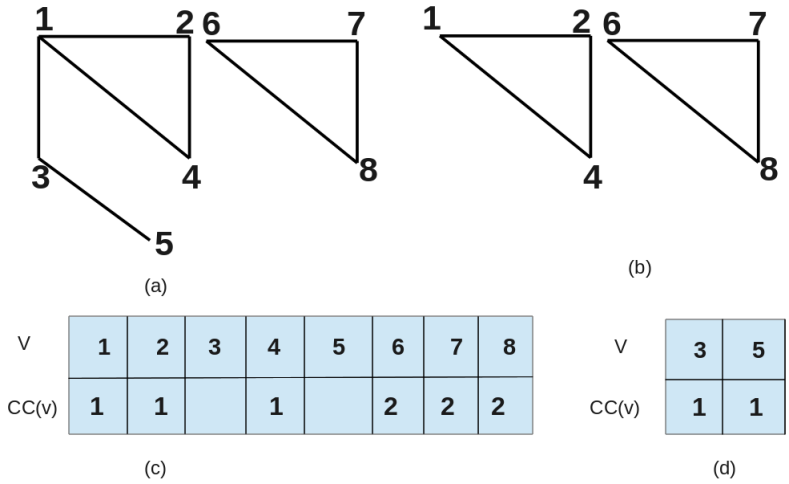

Figure 8.8 An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

for the CPU can be different from that of the GPU. We use the GPU friendly Shiloach-Vishkin (SV) algorithm [119] for the GPU computation and DFS on the individual CPU cores.

GPU and CPU Optimizations : The Shiloach–Vishkin [119] algorithm is a well suited algorithm for parallel implementation. However, we need to make some additional optimizations in order to make it more efficient for the hybrid platform. Some of the major bottlenecks that we address are that of the atomics, memory latencies and reducing divergence. Towards implementing these modifications for the GPU, we perform the SV algorithm in three steps. In the first step, called the *hooking* step, we hook subtrees of the graph whose root has a lower label to a tree with a higher label whenever there is an edge uv such that u and v are in different subtrees. In the second step, we do *pointer jumping* where by we shrink the existing trees to a rooted star so that the nodes are present only at two levels: either at the root level or at the leaf level. In the final step we perform *edge hiding*. In this step, we stop processing the edges of the smaller sub trees once their hooking step is over. This reduces the thread divergence to a great extent and also reduces data movement. More details of these optimizations are described in [123].

For finding the connected components of the graph G_i for $1 \leq i \leq c$ on the ith core of the CPU, we use the standard DFS algorithm [28]. This is motivated by the fact that since the available parallelism on the CPU is small, highly data parallel algorithms do not make a good fit. Further, each CPU core can run independently minimizing any overheads in synchronization and communication. The output of this step is that each CPU core labels the components identified uniquely.

In the final step, Step of Algorithm 19, the following post-processing is done. For a pendant vertex v removed in the ith iteration, let w be the only neighbor of v. Then the vertex v is said to belong to the component that w belongs to. In the above, we process vertices in the opposite order of their removal in Step 1. An example run of our algorithm is presented in Figure 8.8 for the graph from Figure 8.2.

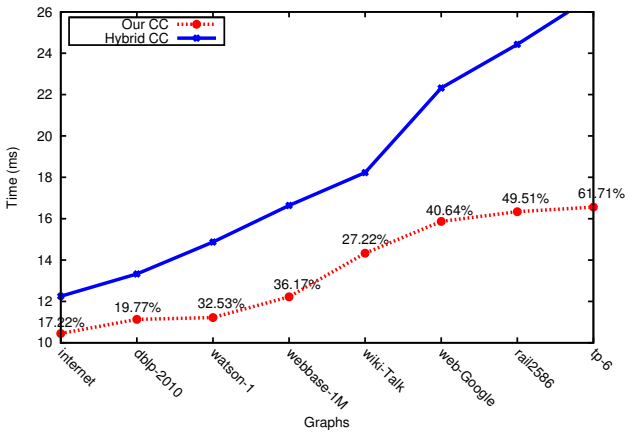

Figure 8.9 Performance of Connected Components the UF Sparse Matrix dataset [2]. The numbers show the percentage improvement.

8.4.2 Results

In this section, we present the results of our implementation. We compare the results with those of [16]. The results of [16] are currently the best reported results for finding the connected components of a given graph on a CPU+GPU platform.

In Figure 8.9, we ran our implementation on a sample of eight real world graphs from the University of Florida dataset [2]. This can be attributed to the fact that the pointer jumping operation in the Shiloach-Vishkin algorithm is a highly irregular operation, and hence the fewer such operations the better.

Similar to the BFS implementation, we also study the the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 8.10 for the graphs webbase_1M, wiki-Talk and Web_Google from the dataset of [2]. Due to the irregular nature of memory accesses, pointer jumping is the predominantly costly operation in the Shiloach-Vishkin [119] algorithm on the GPU. Since our technique reduces the number of these operations, there is a significant impact on the overall runtime.

Finally, we also study the trade-off between the number of iterations of Phase I and the overall runtime. This study is motivated by similar reasons as explained in Section 8.3.5. Figure 8.11 shows the results of the above experiment for the graph webbase_1M from the dataset of [2]. It can be noticed that the overall time taken decreases over iterations of Phase I and plateaus off at about five iterations for the graph under consideration. The time for Phase I is shown on the right-side Y-axis of Figure 8.11.

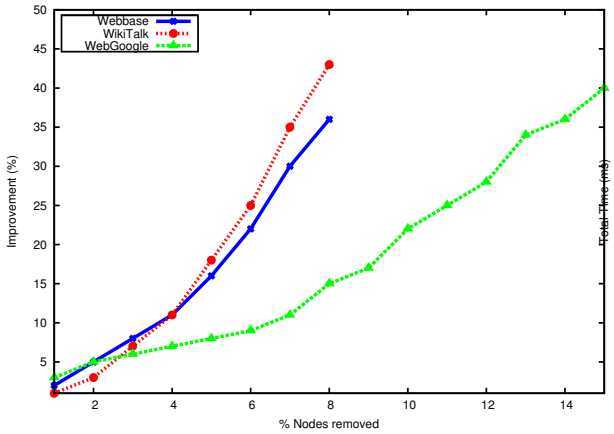

Figure 8.10 Performance improvement of Connected Components with the removal of pendant nodes.

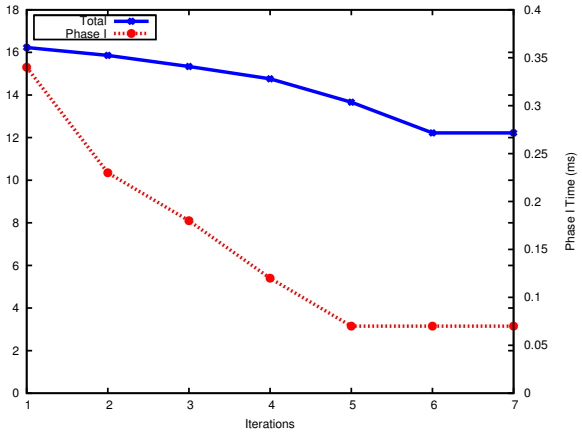

Figure 8.11 Trade-off between Phase I and the total time.

8.5 Conclusions

In this Chapter, we have proposed graph pruning as a technique to speed-up large graph algorithms on modern parallel architectures. We applied the technique to two important problems in graphs. Our results indicate that the technique is quite useful, especially for large sparse graphs.

In all the applications we studied in this Chapter, we needed to prune the pendant vertices. In future, we wish to study other problems that will lead to the discovery of other pruning strategies.

Chapter 9

Conclusions and Future Directions

9.1 Conclusions

Programming models for hybrid platforms is a vital area of research. The whole accelerator based parallel computing community is often looking to develop devices which are having good programmability. In [19], Blelloch proposes a parallel model of computation which is a successor to the previous PRAM model. In this VRAM model of computation, the author shows the use of vector multiprocessors to compute several fundamental computing primitives. It is important to note the relevance of this work as the current generation multiprocessors are all having SIMD and vector processing units which requires vector instructions for performance. In [19], the author shows how a group of primitives can be efficiently used to design a parallel application. The grouping of several primitives as a linear sequence of parallel kernels, provide a seamless approach to designing parallel programs.

It is also of interest to extend our work to systems that has multiple GPUs and CPUs. A platform consisting of Nvidia Tesla coupled with a Intel Xeon server that contains a dual socket or quad-socket CPU will be a more mature platform to experiment upon. In the near future, it is also possible that we might be having communication links that can provide bi-directional data transfer at the same instant of time. Real life examples could be on-die hybrid systems. Such platforms will solve many design challenges for hybrid computations but will equally pose some new ones.

As per the question of implementation of primitives. like the ones that are analyzed in this thesis, there are some few interesting solutions that can be provided for efficient implementation on platforms containing multiple GPUs and CPUs. We can see that, as we perform a static partitioning of the graph, it can be very easily transformed into a problem where we fix a higher number of partitions based on the total number of devices that are available on the platform. For example, we can perform a hierarchical partitioning, where we first decide on a partitioning percentage for the GPU. This percentage is furthur subdivided by the number of GPUs that are available on board. Naturally, the rest of the graph is distributed on the CPU in a similar fashion. There might be another possibility where we might have different GPUs on the same platform which is quite often the case due to the release of new microar-

chitectures within very small span of time. In that case, the partition allocated to for the GPUs can be divided based on the ratio of the total number of cores available on each GPU.

Another point of interest is the implementation in platforms when the GPU global memory is too short to accomodate the total amount of data that is present for the application. While this might be a common problem that is faced by several application developers widely, there are several efficient ways of overcoming this. One is asynchronous memory transfers. It is a very popular way of loading data into the GPU memory using asynchronous transfers that is provided by CUDA through the use of streams. Also, with other programming features such as pinned memory and universal addressing that is provided CUDA 6.0 onwards makes this problem much more easier to tackle. Even in cases where the devices work on non-disjoint data or input that is not statically input into the system, can make very efficient use of pinned memory or universal addressing in order to perform efficient implementations. In such cases, it is often a good idea to design the solutions in a way where the two devices approach the shared data from two different dimensions until they overlap and the algorithm terminates.

In this thesis we attempt to provide efficient hybrid solutions for tightly coupled systems. In this work, we provide high performance algorithms for some semi-numerical computations and does refinement of novel algorithms on the subject. In most parts of this work, we concentrate towards performance gains and efficiency. However, according to recent trends power and monetary expectations are exceeding the considerable limits [57]. Hence all the algorithms are needed to be re-thought with power and energy parameters taken into consideration. These considerations will lead to a healthy development of hybrid computations and heterogeneous computations in general.

9.2 Future Directions

There are several interesting areas which can be investigated during the course of this work. As hybrid computing is a relatively new topic in the world of parallel processing, there are several topics which can be re-looked for finding answers in the hybrid concept. Some of the interesting topics are discussed below.

9.2.1 All Pairs Shortest Path

The All Pairs Shortest Path (APSP) problem is a well known problem in graph theory. There has been some research done on the APSP problem on accelerators such as the one in [134, 69]. As a future work, we plan upon implementing a hybrid algorithm for APSP. We plan upon extending the work on graph pruning in order to shorten the size of very large sparse graphs. It is intuitively understandable that for any sparse node that is removed, the path to it will always pass through its predecessors. Hence, the problem can be well suited to be solved using this technique.

Traditionally, the APSP problem is solved using the Floyd-Warshall algorithm between all the possible pairs of nodes. It is a $O(n^3)$ solution and is ideally not suited for parallel implementations. However,

with the elaborate caching mechanisms that are available in the modern generation multicore processors, it is possible to implement a blocked Floyd-Warshall APSP. It reduces the memory access latencies to a higher degree and provides an optimal solution. Such a solution has been already provided by Venkatraman et al. in [134]. A heterogeneous implementation of the blocked APSP problem has also been shown by Matsumoto et al. in [85]. However, the solution uses a non-overlapped APSP algorithm. In order to achieve a better performance, we intend to extend this solution by doing graph pruning and then performing blocked APSP on CPU and GPU in an overlapped fashion.

9.2.2 Randomized Approaches

Randomized approaches is parallel computing is also an interesting area of research. We plan on deploying the random number generator which we have devised for Monte Carlo simulations. These simulations usually require a good and reliable source of randomness. Randomized algorithms have been proved to perform than deterministic ones in many problems of computer science [90]. High quality random number source is essential for designing cryptographic applications. These applications are of high computational intensity and often require good systems knowledge to provide good hardware implementations. It will be interesting to see if the randomized mechanism perform better in a hybrid setting rather than the homogeneous solutions.

Massively multithreaded computing has entirely changed the horizon of parallel computing and through this thesis we want to further this idea by positioning hybrid computing as the future of parallel computing.

Related Publications

1. Dip Sankar Banerjee, Parkishit Sakurikar, Kishore Kothapalli. **Comparison sorting on hybrid multicore architectures for fixed and variable length keys** at *International Journal of High Performance Computing Applications*. doi: 10.1177/1094342014526906

2. Dip Sankar Banerjee, Shashank Sharma Kishore Kothapalli. **Work Efficient Parallel Algorithms for Large Graph Exploration on Emerging Heterogeneous Architectures** accepted at *Journal of Parallel and Distributed Computing*.

3. Dip Sankar Banerjee, Shashank Sharma, Kishore Kothapalli.**Work Efficient Parallel Algorithms for Large Graph Exploration** at IEEE International Conference on High Performance Computing (HiPC), Bangalore 2013.

4. Dip Sankar Banerjee, Parkishit Sakurikar, Kishore Kothapalli **Fast Parallel Comparison Sorting on Hybrid Multicore Architectures** at *Workshop On Accelerators for Hybrid Exascale Systems (AsHES), held in conjunction with IEEE International Parallel and Distributed Symposium (IPDPS) Boston, 2013*.

5. Dip Sankar Banerjee, Aman Bahl, Kishore Kothapalli.**On Demand Fast Parallel Pseudo Random Number Generator with Applications** at *Workshop On Large Scale Parallel Processing (LSPP), held in conjunction with IEEE International Parallel and Distributed Symposium (IPDPS) Shanghai, 2012* .

6. Dip Sankar Banerjee, Kishore Kothapalli. **Hybrid Algorithms for List Ranking and Graph Connected Components** at *IEEE International Conference on High Performance Computing (HiPC) Bangalore, 2011*.

Bibliography

[1] OpenMP Application Program Interface - Version 4.0 - RC2 - March 2013 . http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf.

[2] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[3] V. Agarwal, F. P. D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of ACM SC*, page 111, 10.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, SPAA '89, pages 11–21, New York, NY, USA, 1989. ACM.

[5] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theory of Computer Science*, 71(1):3–28, Mar. 1990.

[6] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. Technical Report Computer Science Technical Report, ICL-UT-10-04, University of Tennessee, 2010.

[7] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of Biomedical Optics*, 13(6), 2008.

[8] R. J. Anderson and G. L. Miller. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[10] B. Awerbuch and Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, Oct. 1987.

[11] D. A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IEEE IPDPS*, pages 1–10, 2007.

[12] D. A. BADER and G. CONG. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994 – 1006, 2005.

[13] D. A. Bader and K. Madduri. GTgrpah: A suite of synthetic graph generators.

[14] S. Bandyopadhyay and S. Sahni. GRS-GPU radix sort for multifield records. In *International Conference on High Performance Computing (HiPC), 2010*, pages 1 –10, Dec. 2010.

[15] S. Bandyopadhyay and S. Sahni. Sorting large records on a Cell Broadband Engine. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 939 –944, June 2010.

[16] D. S. Banerjee and K. Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. of IEEE 18th Annual International Conference on High Performance Computing (HiPC)*, 2011.

[17] D. S. Banerjee, S. Sharma, and K. Kothapalli. Work Efcient Parallel Algorithms for Large Graph Exploration. In *Proc. of IEEE 20th Annual International Conference on High Performance Computing (HiPC)*, 2013.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[19] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[20] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[21] D. Boas, J. Culver, J. Stott, and A. Dunn. Three dimensional monte carlo code for photon migration through complex heterogeneous media including the adult human head. *Opt. Express*, 10:159–170, Feb 2002.

[22] B. Bollobás. *Random graphs*. Cambridge University Press, 2001.

[23] D. Cederman and P. Tsigas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[24] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[25] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389, 2012.

[26] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, Aug. 1988.

[27] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix sums and List Ranking. *Information and Computation*, 81(3):334–352, 1989.

[28] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms, 2001.

[29] N. Corporation. Cuda: Compute Unified Device Architecture programming guide. Technical report, 2007.

[30] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[31] R. B. D'agostino and M. S. Stephen. *Goodness-of-Fit Techniques*. Marcel Dekker, 1986.

[32] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient Parallel Merge Sort for Fixed and Variable Length Keys. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[33] J. Dongarra. http://www.nvidia.com/object/GPU_Computing.html.

[34] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[35] R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, Feb. 1990.

[36] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Trans. Parallel Distrib. Syst.*, 7(8):791–805, Aug. 1996.

[37] G. DAngelo, M. DEmidio, D. Frigioni, and V. Maurizio. A speed-up technique for distributed shortest paths computation. In *Computational Science and Its Applications-ICCSA 2011*, pages 578–593. 2011.

[38] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[39] O. Fialka and M. Cadik. FFT and Convolution Performance in Filtering on GPU. In *Proceedings of the Conference on Information Visualization*, 2006.

[40] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[41] D. Frank and H. Zaboli. Deterministic Sample Sort For GPUs . *CoRR http://arxiv.org/pdf/1002.4464.pdf*, 2010.

[42] O. Gabber and Z. Galil. Explicit constructions of linear size superconcentrators. In *in IEEE FOCS*, pages 364–370, 1979.

[43] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *in Proc. of IEEE IPDPS*, 2013.

[44] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM.

[45] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[46] N. Govindaraju and D. Manocha. Cache-efficient Numerical Algorithms using Graphics Hardware. *Parallel Computing*, 33(10-11):663–684, 2007.

[47] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, Oct. 2005.

[48] J. Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 16–25. ACM, 1994.

[49] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC*, 2007.

[50] Z. He and B. Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, April 2010.

[51] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. *J. Parallel Distrib. Comput.*, 52(1):1–23, July 1998.

[52] D. R. Helman and J. JàJà. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. In *Proc. ALENEX*, pages 37–56, 1999.

[53] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[54] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, STOC '76, pages 55–57, New York, NY, USA, 1976. ACM.

[55] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22:461–464, August 1979.

[56] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–331, July 1961.

[57] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[58] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration for multi-core CPU and GPU. In *IEEE Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[59] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S)*, 43:439–561, 2006.

[60] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel Implementation of Algorithms for Finding Connected Components in Graphs. *Parallel algorithms: Third DIMACS implementation challenge*, pages 11–20, 1994.

[61] J. Huang and Y. Chow. Parallel sorting and data partitioning by sampling. *7th Computer Software and Applications Conference*, pages 627–631, 1983.

[62] W.-m. Hwu, K. Keutzer, and T. G. Mattson. The concurrency challenge. *IEEE Des. Test*, 25(4):312–320, July 2008.

[63] E. D. B. III and K. H. Warren. The 1991 MPCI yearly report: The attack of the killer micros. Technical Report UCRL-ID-107022, Lawrence Livermore National Laboratory, March 1991.

[64] J. Jaja. *An Introduction To Parallel Algorithms*. Addison-Wesley, 2004.

[65] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July 2005.

[66] H. J. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *J. ACM*, 40:454–476, July 1993.

[67] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the twenty-fourth annual ACM Symposium on Theory of Computing*, STOC '92, pages 318–326, New York, NY, USA, 1992. ACM.

[68] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[69] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55. Eurographics Association, 2008.

[70] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[71] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[72] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA, 1997.

[73] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. 1998.

[74] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The high performance Fortran handbook*. MIT Press, Cambridge, MA, USA, 1994.

[75] Kongetira, Poonacha and Aingaran, Kathirgamar and Olukotun, Kunle. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March 2005.

[76] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *International Conference on High Performance Computing (HiPC)*, pages 463–472, 2009.

[77] P. L'Ecuyer. Efficient and portable combined random number generators. *Commun. ACM*, 31:742–751, June 1988.

[78] P. L'Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33, 2007.

[79] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10. ACM, 2010.

[80] D. Lehmer. Mathematical methods in large scale computing units. In *Proc. 2nd Symp. on Large Scale Digital Comp. Mach.*, pages 141–146, 1951.

[81] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–10. IEEE, 2010.

[82] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A scalable high performance cholesky factorization for multicore with gpu accelerators. In *VECPAR '10: 9th International Meeting on High Performance Computing for Computational Science*, 2010.

[83] G. Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. http://www.stat.fsu.edu/pub/diehard/, 1995.

[84] K. K. Matam and K. Kothapalli. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 612–621, Washington, DC, USA, 2011. IEEE Computer Society.

[85] K. Matsumoto, N. Nakasato, and S. Sedukhin. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. In *2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, pages 145–152, 2011.

[86] M. Matsumoto and T. Nishimura. Dynamic Creation of Pseudorandom Number Generators. In *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pages 56–69, June 1998.

[87] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[88] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, Nov. 1984.

[89] D. G. Merrill and A. S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[90] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[91] L.-M. Munguia, D. A. Bader, and E. Ayguadé. Task-based parallel breadth-first search in heterogeneous environments. In *HiPC*, pages 1–10, 2012.

[92] D. R. Musser. Introspective sorting and selection algorithms. *Software Practise and Experience*, 27(8):983–993, Aug. 1997.

[93] J. V. Neumann. Various techniques used in connection with random digits. *Applied Mathematics Serires*, 12:36–38, 1951.

[94] B. Nichols, D. Buttlar, and J. Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. A Nutshell handbook. O'Reilly Media, Incorporated, 1996.

[95] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[96] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[97] NVIDIA Corporation. CUDA CURAND Library, Aug. 2010.

[98] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, Sept. 2005.

[99] M. A. Overton. http://drdobbs.com/tools/229625477.

[100] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[101] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the twentieth annual ACM Symposium on Theory of computing*, STOC '88, pages 510–513, New York, NY, USA, 1988. ACM.

[102] S. Patidar and P. J. Narayanan. Scalable split and sort primitives using ordered atomic operations on the gpu. In *High Performance Graphics*, April 2009.

[103] B. Pattabiraman, M. M. A. Patwary, A. H. Gebremedhin, W. keng Liao, and A. N. Choudhary. Fast algorithms for the maximum clique problem on massive sparse graphs. *CoRR*, abs/1209.5818, 2012.

[104] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place sorting with CUDA based on bitonic sort. In *Concurrency and Computation: Practice and Experience*, pages 681–693, 2011.

[105] V. Podlozhnyuk. Parallel mersenne twister. Technical report, 2007.

[106] V. Podlozhnyuk. Histogram calculation in CUDA. *CUDA http://developer.download.nvidia.com/compute/cuda/histogram.pdf*, 2009.

[107] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[108] M. S. Rehman, K. Kothapalli, and P. J. Narayanan. Fast and Scalable List Ranking on the GPU. In *Proc. of ACM ICS*, 2009.

[109] M. Reid-Miller. List Ranking and List Scan on the Cray C-90. In *Proceedings of the Sixth Annual ACM symposium on Parallel Algorithms and Architectures*, pages 104–113, 1994.

[110] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[111] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs, GPUs and Intel MIC Architectures. Technical report, Intel Labs, 2010.

[112] D. P. Scarpazza, O. Villa, and F. Petrini. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, 2008.

[113] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 33–37, New York, NY, USA, 2007. ACM.

[114] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, Dec. 1988.

[115] L. R. Scott, T. Clark, and B. Bagheri. *Scientic Parallel Computing*. Princeton University Press, Princeton, NJ, USA, 2005.

[116] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2007.

[117] J. Shalf, M. Wehner, and L. Oliker. The challenge of energy ecient HPC. Scidac review, (14):5057, 2009, 2009.

[118] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 39:1–39:11, New York, NY, USA, 2009. ACM.

[119] Y. Shiloach and U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms*, pages 57–67, 1982.

[120] J. Sibeyn. Minimizing Global Communication in Parallel List Ranking. In *Proceedings of Euro-Par Parallel Processing*, pages 894–902, 2003.

[121] J. Singler, P. Sanders, and F. Putze. MCSTL: the multi-core standard template library. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 682–694, Berlin, Heidelberg, 2007. Springer-Verlag.

[122] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, Oct. 2008.

[123] J. Soman, K. Kothapalli, and P. Narayanan. Some GPU Algorithms for Graph Connected Components and Spanning Tree. In *Parallel Processing Letters*, volume 20, pages 325–339, 2010.

[124] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7), Sept. 2005.

[125] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 14–19, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[126] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense liner algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 12:10–16, Dec. 2009.

[127] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[128] S. Tomov, H. Ltaief, R. Nath, and J. Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proc. of IPDPS*, 2010.

[129] S. Tomov, H. Ltaief, R. Nath, and J. Dongarra. Faster, Cheaper, Better - A Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*. 2010.

[130] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 2010.

[131] S. Tzeng and L.-Y. Wei. Parallel white noise generation on a gpu via cryptographic hash. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 79–87, New York, NY, USA, 2008. ACM.

[132] UniProt. `http://www.uniprot.org/`.

[133] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8):103 – 111, 1990.

[134] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics*, 8, Dec. 2003.

[135] O. Villa, D. Scarpazza, F. Petrini, and J. Peinador. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[136] V. Vineet, P. Harish, and P. J. Narayanan. Fast minimum spanning tree for large graphs on gpu. In *Proceedings of High Performance Graphics*, 2009.

[137] V. Vineet and P. J. Narayanan. CUDA Cuts: Fast Graph Cuts on the GPU. In *Proceedings of the CVPR Workshop on Visual Computer Vision on GPUs*, 2008.

[138] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, Oct. 1993.

[139] Z. Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[140] T. White and J. Dongarra. Overlapping computation and communication for advection on a hybrid parallel computer. In *Proc. of IPDPS*, May 2011.

[141] J. C. Wyllie. *The complexity of parallel computations*. PhD thesis, Cornell University, Ithaca, NY, 1979.

[142] Y. Xia and V. K. Prasanna. Topologically Adaptive Parallel Breadth Frst Search on Multicore-Processors. In *in Proc. PDCS*, 2009.

[143] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.

[144] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley & Sons, 2004.

[145] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 712–721, New York, NY, USA, 1991. ACM.