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.
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 \texttt{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-
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 Helman-Jaja algorithm for the GPU which can rank 32 million elements in a second and 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 their 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](image.png)

**Figure 3.4** Improvement of List Ranking from 2009 to the current hybrid implementation. We can see that we achieved a speedup of nearly 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 $O(n \log n)$ 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](image)

**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](image-url)

**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].