Hybrid Algorithms for List Ranking and Graph Connected Components

Dip Sankar Banerjee and Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
{dipsankar.banerjee@research.,kkishore@}iiit.ac.in

Abstract— **The advent of multicore and many-core architectures saw them being deployed to speed-up computations across several disciplines and application areas. Prominent examples include semi-numerical algorithms such as sorting, graph algorithms, image processing, scientific computations, and the like. In particular, using GPUs for general purpose computations has attracted a lot of attention given that GPUs can deliver more than one TFLOP of computing power at very low prices.**

In this work, we use a new model of multicore computing called *hybrid multicore computing* where the computation is performed simultaneously a control device, such as a CPU, and an accelerator such as a GPU. To this end, we use two case studies to 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 paper 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 [Z. Wei, J. JaJa; IPDPS 2010]. Similarly, our hybrid algorithm for graph connected components is faster by 25% compared to the best known GPU implementation [26].

Keywords: **Hybrid multicore algorithms, list ranking, connected components, GPGPU.**

I. Introduction

Modern computing is presently undergoing a power and performance driven change. This is leading to the development of more on-die cores. The current architecture space is populated by homogeneous systems such as the eight-core processors and the development trend indicates that 16 and 32 core processors would soon be a reality. In recent years, accelerator-based computing using accelerators such as the IBM Cell SPUs, FPGAs, GPUs, ASICs is studied with clear performance gains compared to CPUs. Of these, GPUs stand out in a unique way from all these innovative solutions because they are produced as commodity processors. GPUs though are made mostly for graphics and image processing are nowadays making a big headway as general purpose processors because of their superior computing power, low energy usage, and low cost.

It is observed that GPUs are good at improving the performance of regular computations such as those described in [23], [9], [6]. 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 [33], [21], connected components [26], 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 [16], 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.

The above suggests that as also 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. In particular, Jack Dongarra [8] quotes:

> 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.

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. 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.

A. 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

978-1-4577-1950-9/11/$26.00 ©2011 IEEE

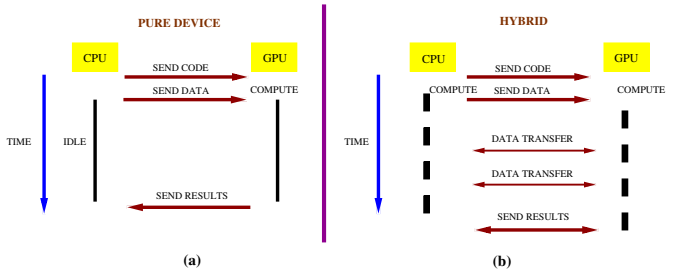

Fig. 1. The case for hybrid computing. (a) A pure device execution mechanism. (b) Hybrid execution mechanism.

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 [23], [33], [21]. As can be seen in Figure 1(a), 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 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.
- **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, as shown in Figure 1(b), 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 [34], [31], [11]. 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 model, Tesla S1070, and other leading GPU platforms, allows kernel calls to be executed in a non-blocking manner[19]. 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.

B. Related Work

General purpose computing on the GPUs, termed GPGPU, has matured enough in the research community. Some of the important works include [9], [26], [33]. In most of these works, the computation is entirely performed on the GPU. In this paper, we do not wish to provide a complete review of these works.

In a recent work, Lee et al. [16] 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 [16] 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 [31]. In [31], 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 [1], Cholesky factorization [17], triangular forms [30], and other associated works [28], [29], [27]. These works justify the applicability of hybrid computing platform for a variety of dense linear algebra problems.

In [11], 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 [33] 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 [12]. The Helman-JaJa solution has been used in the work of [3] on the Cell, in the work of [21] and [33] on the GPUs. At present, the results of [33] 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., [13], [24]. 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 [10], 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 [26].

C. Our Results

In this paper, we use the hybrid computing platform as described in Section II-C. 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[12]. 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 II-C 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 [33]. 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 [24], 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 [26]. We also notice that our hybrid algorithm has very minimal idle time. We also show that our approach can lead to auto-tuning.

D. Organization of the Paper

The rest of the paper is organized as follows. In Section II, we describe our hybrid computing platform. Section III discusses our first case study using list ranking as the problem. Section IV discusses the second case study using the graph connected components as the problem. The paper ends with a few concluding remarks in Section V.

II. Preliminaries

A. A Brief Overview of NVidia GPUs

Nvidia's unified architecture (see also the left half of Figure 2) for its current line of GPUs supports both graphics and general computing. In general purpose computing, the GPU is viewed as a massively multi-threaded architecture containing hundreds of processing elements (*cores*). Each core comes with a four stage pipeline. Eight cores, also known as *Symmetric Processors* (SPs) are grouped in an SIMD fashion into a *Symmetric Multiprocessor* (SM), so that each core in an SM executes the same instruction. The Tesla C1060 has 30 such SMs, which makes for a total of 240 processing cores. Each core can store a number of thread contexts. Data fetch latencies are tolerated by switching between threads. Nvidia features a zero-overhead scheduling system by quick switching of thread contexts in the hardware.

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and blocks make up a *grid*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA has a zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp.

The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. 16 Kilobyte of *shared memory* per SM acts as a user-managed cache and is available for all the threads in a Block. The Tesla C1060 is equipped with 4 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Prior to launching a kernel, all the data required for the computation must be transferred from the *host* (CPU) memory to the GPU *global* memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. For more details, we refer the interested reader to [20], [18].

B. The Multicore CPU

The Intel i7 980 CPU that we use in our experiments is a recent six core offering from Intel. It is shown in the right half of Figure 2. This processor is the latest 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 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

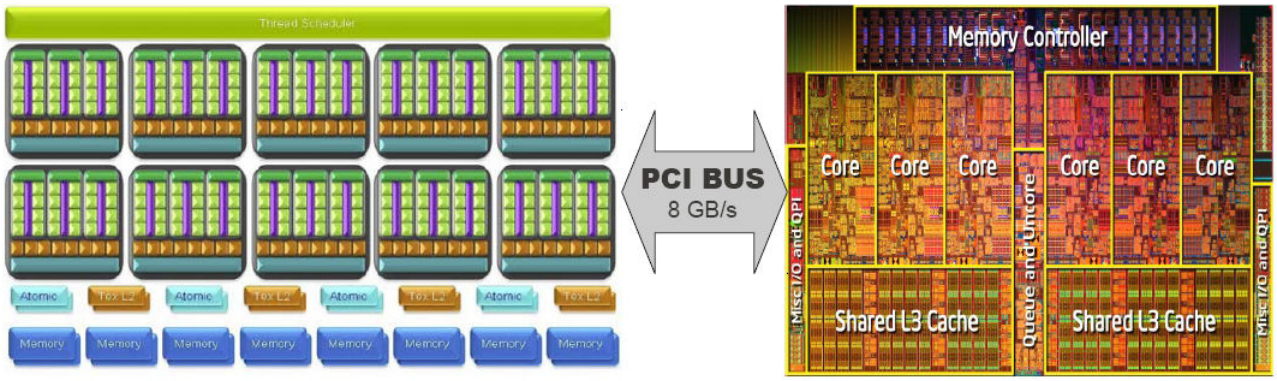

Fig. 2. The GPU-CPU hybrid platform. The pictures are obtained from $http://www.hothardware.com$

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. The memory bandwidth is up to 1066 MHz. The Intel i7 980 CPU has a peak double precision throughput of about 100 GFLOPS.

C. Our Hybrid Platform

Our hybrid platform as shown in Figure 2 is a coupling of the two devices described above, the Intel i7 980 and the Nvidia Tesla C1060 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 3.2 [19]. For programming the CPU, we use OpenMP specification 3.0 and ANSI C [15]. The CUDA API Version 3.2 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. Asynchronous transfer of data from the CPU to the GPU is also supported through *s*treams. This facility allows one to overlap not only executions on the CPU and the GPU but also data transfers between the CPU and the GPU.

III. 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 [35] 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 [2], [35], [12], [25], [22].

Given the importance of the problem, recently solutions optimized to emerging architectures such as the GPU [21], [33], and the IBM Cell [3] have been reported. The solution of [21] uses only the GPU to perform list ranking. The solution of [33] uses the CPU also but not in the manner we envisage in hybrid multicore computing. In [33], 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.

A. 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 1.

Algorithm 1 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 [14]. 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.

B. 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. [14]) that with high probability, the FIS constructed above has at least n/c nodes for $c \geq 24$.

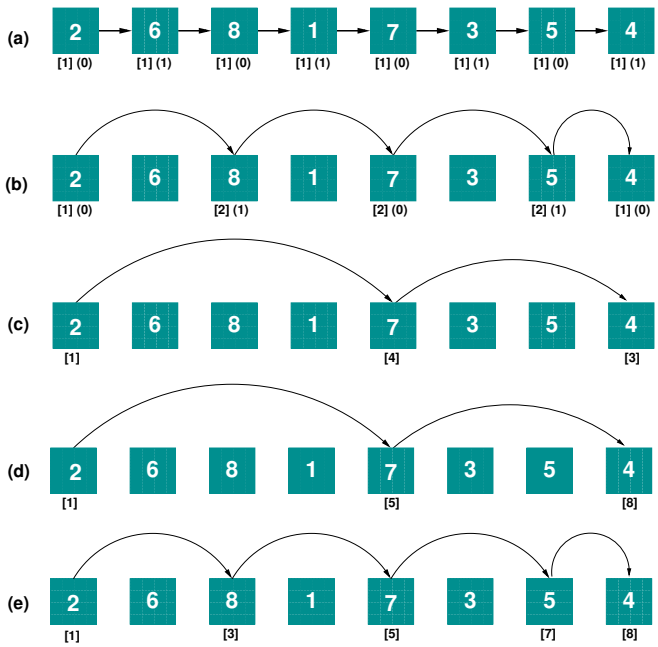

Fig. 3. The linked list and pre-processing done. (a) The initial list with the rank values in square brackets and random string in parenthesis. (b) Elements removed on the basis of the random string and ranks adjusted (c) List obtained after the pre-processing, (d) Ranking the remaining list, and (e) Restoration of the nodes removed from the list.

Our complete algorithm is shown in Algorithm 2. An example run of the algorithm is shown in Figure 3. We discuss each phase of the algorithm in detail as follows.

Phase I Issues: Recent works on list ranking [21], [33], [3], 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. [14]). 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 III-C. The number of iterations r will be discussed later in Section III-C.

Algorithm 2 LISTRANK($List\ L$)

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

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

In Algorithm 2, 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. 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 2 is justified in the following section. Lines 5–11 of Algorithm 2 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 [21], 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 [33]. 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 [14].

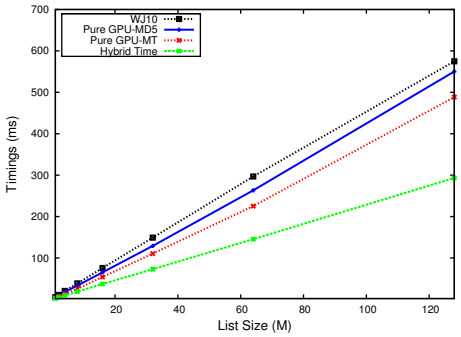

Fig. 4. Time Comparison with respect to pure GPU implementations and the best known result.

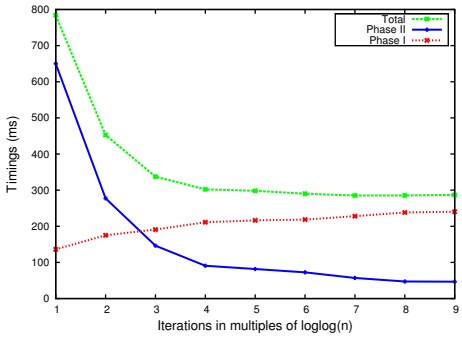

Fig. 5. Trade offs between Phase I and Phase II and the total timings for a 128M sized list

C. Experimental Results

The experimental platform which we use has been already described in II-C. 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 2 with the following alternative approaches. The algorithm presented in [33] 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 [33]. We also run the algorithm presented in Algorithm 2 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 [32]. A recent work [20] uses the Mersenne Twister based approach to generate random numbers on the GPU. Both these two methods are used for experimenting on Algorithm 2.

Figure 4 shows the runtime of our algorithm on lists of various sizes in comparison with the timings from Wei and JaJa [33]. In Figure 4, the label "Hybrid time" refers to our approach, and the label "WJ10" refers to the timings from [33]. 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 [20].

It can be observed from Figure 4 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 [33] 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 2 on the GPU alone, by generating random numbers also on the GPU, improves the results of [33] 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.

In the plot labeled "WJ10", we use the numbers reported in [33]. Since the GPU hardware used by [33] 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 [33]. 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 2, 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 5 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 4, the values reported correspond to the above value of r.

In Table I 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 [33]. 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 [33], these results will also improve the corresponding results from [21], [3], [22].

To showcase the power of our hybrid algorithm approach, we focus on Phase I of our algorithm and illustrate the runtime

TABLE I

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

TABLE II

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

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 II. The first row denotes the first generation and transfer. It can be noticed from Table II 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.

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 where the time taken by each step of Phase I is shown for a list of 128 M elements. Figure 6 shows also how the various steps in Phase I overlap in their execution. It can be indeed noticed from Figure 6 that the GPU is idle only for less than 5% of the time.

IV. 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.

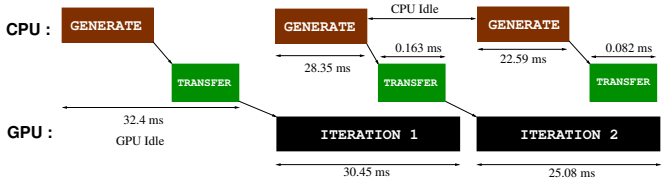

Fig. 6. Work units in overlapped execution for a list of 128 M elements for the first 2 iterations.

- 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 3 and [26]. A brief pseudo-code of our approach is given in Algorithm 3. In Sections IV-A– IV-C, we describe each step of the algorithm in detail.

Algorithm 3 CONNECTED COMPONENTS(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 $1 \le i \le c$, $V(G_{c+1}) = V(G) - \cup_{i=1}^{c} V(G_i)$, $E(G_i) = E(G) \cap (V_i \times V_i)$, for $1 \le i \le c+1$.

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, $1 \le i \le c$

5: `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 [24] 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 shows an example of cross edges. These cross edges are considered in Algorithm 4 described in Section IV-C.

A. 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 [24].

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 [26]. 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 [26] 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 [26] and presently the results of [26] outperform the other GPU-based connected-component algorithms.

B. 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 [7]. 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.

C. 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.

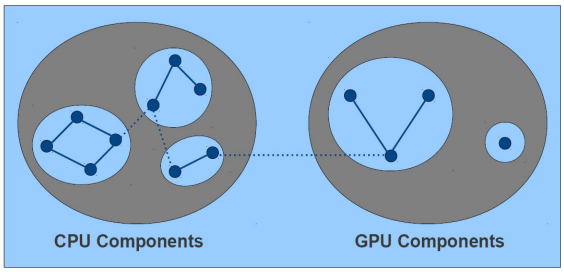

Fig. 7. Super-graph connected by the cross-edges. The cross-edges are denoted by the broken lines.

We now present Algorithm 4 that identifies connected components in the supergraph. In Algorithm 4, 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 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 4 PROCESSCROSSEDGES($Supergraph\ S$)

```
1: while Labels change do
2:   for each of the cross-edges uv where uv ∈ 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
```

D. Results

Our experimental set up is as described in Section II-C. In this work, we consider random graphs of several components. These random graphs are generated according to the $\mathcal{G}(n,p)$ model [5] using the GTgraph generator [4].

In Figure 8 we show the speed-up achieved by our hybrid algorithm on several graph sizes. The label "Hybrid Time" refers to the runtime of Algorithm 3. The label "GPU Time" refers to the time obtained by the code from the work of [26] on the same GPU as is used in our computing platform. Both these implementations are run on the same datasets generated

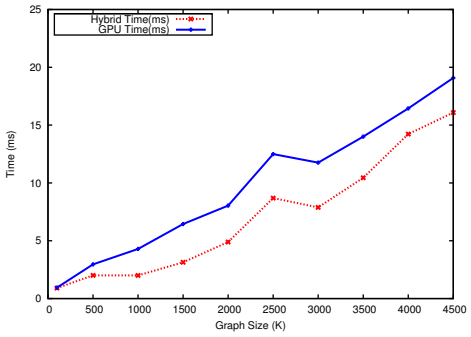

Fig. 8. Time comparison with graphs sizes varying from 500K to 4.5M nodes. The GPU times are obtained by running the code from [26] on the same instances.

as described in the previous paragraph. As can be seen from Figure 8, the hybrid algorithm consistently outperforms the best known GPU implementation by about 25% on average. In all the results presented in Figure 8, and also elsewhere in this section, the experiment is run for multiple runs and the average result is considered. In Figure 8, 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 9 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 9 that the speedup is initially negative and then increases. From Figure 9 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.

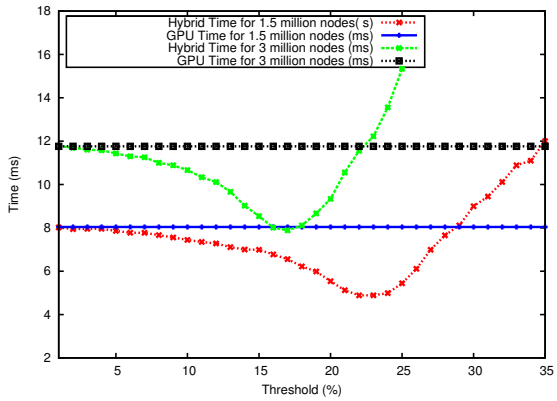

Fig. 9. Time comparison in graphs of size 1.5 million and 3 million nodes. The GPU timings are the obtained by running the code from [26] on the same instances.

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 10. 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 10 where the threshold varies accordingly.

TABLE III
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

It can be noted that the Algorithm 3 when run with a threshold of 0% will correspond to the algorithm of [26]. 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 III 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. It can be seen from Table III 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.

E. 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 9 , for a graph 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 10. One can use the information from Figure 10 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 11.

One aspect to note from Figure 10 and Figure 11 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.

V. CONCLUSIONS AND FUTURE WORK

Our work here considered two fundamental problems and proposed hybrid multicore algorithms for these two problems.

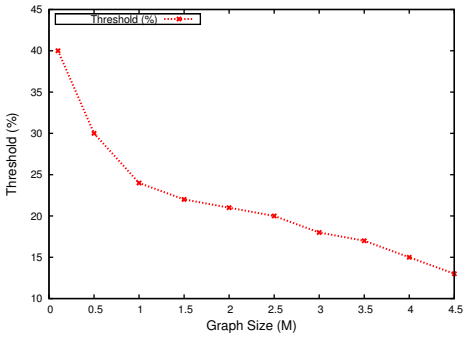

Fig. 10. Threshold variation on graphs with sizes varying from 500K to 4.5M

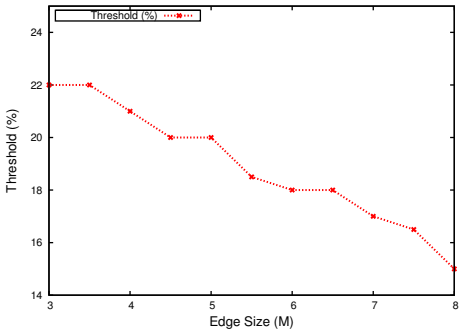

Fig. 11. Threshold variation on a graph of 2.5 million nodes and edge sizes varying from 3M to 8M

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.

REFERENCES

[1] 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," University of Tennessee, Tech. Rep. Computer Science Technical Report, ICL-UT-10-04, 2010.

[2] R. J. Anderson and G. L. Miller, "A Simple Randomized Parallel Algorithm for List-Ranking," *Information Processing Letters*, vol. 33, no. 5, pp. 269–273, 1990.

[3] 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 IPDPS*, 2007, pp. 1–10.

[4] D. A. Bader and K. Madduri, "GTgrpah: A suite of synthetic graph generators." [Online]. Available: https://sdm.lbl.gov/ kamesh/software/GTgraph/

[5] B. Bollobs, *Random graphs*. Cambridge University Press, 2001.

[6] R. Cole and U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking," *Info. and Comput.*, vol. 81, no. 3, pp. 334–352, 1989.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to Algorithms," 2001.

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

[9] N. Govindaraju and D. Manocha, "Cache-efficient Numerical Algorithms using Graphics Hardware," *Parallel Computing*, vol. 33, no. 10-11, pp. 663–684, 2007.

[10] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proc. SPAA*, 1994, pp. 16–25.

[11] Z. He and B. Hong, "Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms," in *Proc. IPDPS*, 2010.

[12] D. R. Helman and J. JàJà, "Designing Practical Efficient Algorithms for Symmetric Multiprocessors," in *Proc. ALENEX*, 1999, pp. 37–56.

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

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

[15] B. W. Kernighan and D. M. Ritchie, *The C Programming Language Second Edition*. Prentice Hall, 1988.

[16] 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 *Proc. ISCA*, 2010.

[17] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," in *Proc. of VECPAR'10*, 2010.

[18] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[19] Nvidia Corporation, "CUDA: Compute Unified Device Architecture programming guide," Technical report, Nvidia, Tech. Rep., 2007.

[20] V. Podlozhnyuk, "Parallel Mersenne Twister," Nvidia, Tech. Rep., 2007.

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

[22] M. Reid-Miller, "List Ranking and List Scan on the Cray C-90," in *Proc. ACM SPAA*, 1994, pp. 104–113.

[23] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. ACM Symp. GH*, 2007.

[24] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm." *J. Algorithms*, pp. 57–67, 1982.

[25] J. Sibeyn, "Minimizing Global Communication in Parallel List Ranking," in *Proc. of Euro-Par.*, 2003, pp. 894–902.

[26] J. Soman, K. Kothapalli, and P. J. Narayanan, "Some GPU Algorithms for Graph Connected Components and Spanning Tree," *Parallel Processing Letters*, vol. 20, no. 4, pp. 325–339, 2010.

[27] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

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

[29] ——, "Faster, Cheaper, Better - A Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, 2010.

[30] 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.

[31] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Liner Algebra for Hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 12, pp. 10–16, Dec. 2009.

[32] S. Tzeng and L.-Y. Wei, "Parallel white noise generation on a GPU via cryptographic hash," in *Proc. of I3D*, 2008, pp. 79–87.

[33] Z. Wei and J. JaJa, "Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA," in *Proc. of IPDPS*, April 2010.

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

[35] J. C. Wyllie, "The complexity of parallel computations," Ph.D. dissertation, Cornell University, Ithaca, NY, 1979.