Lessons Learned from Exploring the Backtracking Paradigm on the GPU

John Jenkins[1,2], Isha Arkatkar[1,2], John D. Owens[3], Alok Choudhary[4], and Nagiza F. Samatova[1,2,5]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN 37830, USA
[3] University of California, Davis, Davis, CA 95616, USA
[4] Northwestern University, Evanston, IL 60208, USA
[5] Corresponding author: samatova@csc.ncsu.edu

Abstract. We explore the backtracking paradigm with properties seen as sub-optimal for GPU architectures, using as a case study the maximal clique enumeration problem, and find that the presence of these properties limit GPU performance to approximately 1.4–2.25 times a single CPU core. The GPU performance "lessons" we find critical to providing this performance include a *coarse*-and-*fine*-grain parallelization of the search space, a low-overhead *load-balanced* distribution of work, global memory latency hiding through *coalescence*, *saturation*, and *shared memory utilization*, and the use of GPU *output buffering* as a solution to irregular workloads and a large solution domain. We also find a strong reliance on an efficient global problem structure representation that bounds any efficiencies gained from these lessons, and discuss the meanings of these results to backtracking problems in general.

1 Introduction

The backtracking paradigm, a depth-first search method that finds solutions in a memory efficient manner, is ubiquitous in computing. A few examples include constraint satisfaction in AI [11], frequent itemset mining in data mining [6], maximal clique enumeration in graph mining [16], k-d tree traversal for ray tracing in graphics [9], and logic programming languages such as Prolog. Backtracking typically constructs optimal solutions from candidate solutions, thus forming a search tree that the backtracking traverses. Backtracking is oftentimes at the core of the problems that are combinatorial by nature and, therefore, compute-and-memory-intensive. For many such problems, performing a breadth-first search of the search tree is infeasible due to memory requirements. For instance, frequent itemset mining, as exhibited by the *Apriori* algorithm and its variants [1], becomes infeasible for large input domains.

To reduce backtracking's computational requirements, several strategies have been explored. Pruning the search tree by eliminating non-candidate subtrees, such as in α-β game-tree pruning, avoids unnecessary computation. Likewise, an efficient data model for problem representation (e.g., bitmaps) enables efficient

use of intermediate data structures. Finally, parallel implementation of backtracking search on HPC multi-node, multi-core architectures offers scalability for large problem domains (e.g., parallel maximal clique enumeration (MCE) in graphs [16]).

Recent advancements in parallel computing architectures have opened up possibilites for more computationally- and energy-efficient algorithms. In particular, graphics processing units (GPUs) have been maturing not only for graphics applications, but also for general-purpose computations[6] [14]. Some computational motifs perform effectively on a GPU, while the effectiveness of others is still an open issue. For instance, Lee et al. note an average speedup of 2.5× of various algorithms on the GPU vs. optimized Nehalem implementations, and both Lee et al. and Vuduc et al. highlight memory-bound algorithms on the GPU that perform at the same level or worse than the corresponding CPU implementation [12, 18].

Despite some of the successes of recent computational dwarfs on GPUs, the mapping of the backtracking paradigm onto the GPU architecture has been recognized as a notoriously difficult problem for a number of reasons. Table 1 names a number of difficulties that a mapping of a backtracking problem to the GPU could encounter, leading to a vastly inefficient use of the GPU memory hierarchy and SIMD-optimized GPU multi-processors.

There have, however, been algorithms successfully mapped onto the GPU, though with major departures from the general case of backtracking. The most visible example is in ray tracing, where k-d tree acceleration structures are used to compute ray intersections by traversing the tree in a depth-first fashion [5, 10]. The tree is computed in full before traversals, and the stack-based representation is eliminated by explicitly computing transitions through the tree. However, these properties cannot be assumed in many backtracking problems, much less in the general case.

Our goal, therefore, is to investigate the parallelization of the backtracking paradigm on the GPU. To do this, we analyze the components of *difficult* backtracking problems and propose *tree-level* and *node-level* parallelizations of search space traversal, as well as *buffer-based* output. At best, given the performance of other computational motifs and the nature of the backtracking problem, we cannot expect an order of magnitude increase in performance. Rather, a more realistic performance goal is to perform at one to two times the CPU performance, which opens up the possibility of building future backtracking algorithms on heterogeneous hardware (such as CPU-GPU clusters) and performing workload-based optimizations.

2 Motivation

As mentioned, backtracking is a depth-first exploration of a problem space, where states represent partial solutions. At each step, either the partial solution is expanded to another possible solution, or it is determined that a solution cannot possibly lie on this path, and the search *backtracks* to a previous state. Some

[6] All further discussion will be based on Nvidia's CUDA architecture.

backtracking problems are harder than others, and it is the characteristics of the harder ones that are of the most interest. Table 1 summarizes these characteristics compared to optimal GPU conditions.

Table 1: Opposing algorithm and hardware characteristics.

	Backtracking	**GPU optimal**
Problem Instance	Irregular access	Regular access with locality
Work Unit	Memory, computation variable	Constant size, perfectly SIMD
Output	Exponential size (if enumerative), hard to estimate	Polynomial size, apriori
Search Space	Tree-based, unbalanced	Fixed, apriori (if applicable)

The problem instance can lend itself to irregular access patterns, making it difficult for GPU algorithms to coalesce memory accesses. One example of this is an adjacency list representation of graphs, where vertices may link to arbitrary other vertices. This problem has been recognized by attempts to perform graph algorithms such as breadth-first search on the GPU [8]. In many cases, graphs are too large to use an adjacency matrix representation.

In many problems, the search node, or work unit, is variable in both memory and computational requirements, making load balancing, enforcing thread convergence, and efficiently utilizing processors and storage mechanisms difficult. One example is an instance of constraint satisfaction, where solutions are subsets of a very large set.

The output size of enumerative problems can be exponential with respect to the problem size. For instance, finding all maximal cliques in a graph has a worst-case exponential output size [13]. This can limit acceleration of a GPU-based method due to overhead in CPU-GPU memory transfers.

Finally, the search tree in many backtracking problems is unpredictable, making it difficult to divide the work evenly. For example, in the context of MCE, current parallel methods rely on communication between compute nodes to load balance and distribute [16], whereas on GPUs thread blocks are optimized to perform independently of each other.

3 Backtracking Case Study: Bron-Kerbosch MCE

3.1 Algorithm Overview

A *clique* of a graph is a subset of the vertex set in which there is an edge connecting each pair of vertices in the set, and a *maximal clique* is a clique that is not contained in any other, larger clique. Maximal clique enumeration (MCE) is ubiquitous in real world problems. Examples of the uses of MCE include identification of common secondary structure elements of proteins [7], detection of protein-protein interaction complexes [19], clustering of similar mass spec-

trometry spectra [17], and detection of social heirarchy from email communications [15]. Thus, efficient MCE algorithms are of high value.

The MCE algorithm by Bron and Kerbosch (BK) employs a backtracking strategy that embodies the properties in Table 1, constructively building *maximal cliques* of an input graph [3]. Each subtree being traversed has a *compsub* list, or a list representing the current clique, and each search node consists of two data structures, collectively known as a *candidate path*:

1. *candidate*—the vertices connected to all vertices in *compsub*: these may be added to *compsub* to create a new clique; and
2. *not*—the vertices connected to all vertices in *compsub* that would create a redundant clique if added.

Procedure 1: enum(`cp_stack`, `compsub`): traverse subtree(s) in `cp_stack`, using global `compsub`. Both CPU and GPU use multiple stacks and split among compute elements to achieve *coarse-grain* parallelism.

```
1 // process-per-stack on CPU, warp-per-stack on GPU
2 while not empty(cp_stack) do
3     cp ← pop(cp_stack)
4     update compsub
5     if empty(not(cp)) and empty(cand(cp)) then
6         output compsub
7     else
8         spawn(cp_stack, cp)
9 // CPU -- steal work from other stacks
10 // GPU -- assign stack on CPU side to split work with
11 load_balance(cp_stack)
12 if not empty(cp_stack) then
13     goto 2
```

Procedures 1 and 2 show the enumeration routine. The variable `cp_stack` is a stack data structure, pushing and popping *candidate path* structures in depth-first fashion, in lieu of a recursive representation of backtracking. The stack(s) are initially populated with size-one cliques, that is, a vertex and its neighbors, where the *not* and *candidate* lists are determined lexicographically by vertex label. Until the stack is empty, a process gets the current *candidate path*, and either outputs its *compsub* in the case of a maximal clique, or iteratively creates new *candidate path*s by choosing a vertex to expand (which is added to *compsub* when the new *candidate path* is visited) and computing new *candidate* and *not* lists based on adjacency to the selected vertex. Search tree pruning is performed by the addition of `fixv`, reducing by a large degree the number of subtrees leading to redundant cliques.

Procedure 2: spawn(cp_stack, cp): expand candidate path cp onto stack cp_stack, the GPU splits the procedures in lines 2, 9, 10, 14 to achieve *fine-grain* parallelism.

```
1 // finding fixv is warp-level parallel on GPU
2 fixv ← minimum disconnected vertex to vertices in cand(cp)
3 if fixv in not(cp) then
4    cv ← first vertex in cand(cp) not adjacent to fixv, or nil
5 else
6    cv ← fixv
7 while cv ≠ nil do
8    // filter(cond_fn, list) is warp-level parallel on GPU
9    not(newcp) ← filter(adjacent_to_fixv, not(cp))
10   cand(newcp) ← filter(adjacent_to_fixv, cand(cp))
11   push(cp_stack,newcp)
12   move cv to not(cp)
13   // finding next cv is warp-level parallel on GPU
14   cv ← next vertex not adjacent to fixv, or nil
15 // CPU -- service load balance requests from other processes
```

3.2 Algorithm Parallelization

On both the CPU and GPU variants of the algorithm, *coarse-grain* parallelization is achieved by performing Procedure 1 for many stacks, partitioned among processes. For the GPU variant of the algorithm, *fine-grain* parallelization is performed on the warp level by performing lines 2, 9, 10, and 14 of Procedure 2 in parallel. To find the minimum disconnected vertex `fixv`, each thread in the warp takes vertices in the *not* and *candidate* list in strides, recording the minimum non-connectivity counts, then the warp performs a prefix-sum-like operation to retrieve the global minimum. For instance, for a *not* and *candidate* list of total size n, thread zero computes the local minimum of vertices at offsets 0, 32, etc. thread one computes the local minimum at offsets 1, 33, etc. until all n vertices have been processed. To perform the `filter` operation, the warp steps through the *not* and *candidate* lists in strides, testing connectivity to `fixv`, and uses a prefix-sum to compute the correct offsets to output connected vertices to `fixv`. To determine the next `cv`, each thread in the warp takes a vertex of the remaining *candidate* in strides, testing connectivity, and performs a prefix-sum-like operation to return the correct vertex. Shared memory is used to store warp-wide variables such as *candidate path* information as well as buffers for performing the prefix-sum operations. All shared memory accesses utilize the broadcasting mechanism, where each thread accesses the same memory bank, and avoid bank conflicts for operations such as the prefix-sum. *candidate path*s are also loaded into shared memory in two ways: partially and in full. The partial load method is used when finding `fixv`, loading the *candidate* and *not* lists in warp-level chunks and iteratively testing connectivity between those vertices and the thread-local vertex. Loading the *candidate path* in full allows performance of all operations

on it in shared memory, at the cost of lower occupancy from increased storage requirements (the *candidate path* is size-bound by the maximum vertex degree).

Unlike CPUs, GPUs do not have the capability of outputting directly to disk, so a more complex method of handling output data must be considered. Furthermore, in enumerative problems such as MCE, it is infeasible to store all output solutions in GPU memory at once, so there must be some intermediate CPU-GPU transfers. Naïvely, each stack's *compsub* could be transferred after each expansion iteration to the CPU, where the valid solutions are extracted and output. However, such a method would suffer from low density of usable output. The more efficient way is to use atomic operations to reserve space from a pre-allocated output buffer and allow blocks to continue expanding states until the buffer is full, decoupling the strict expand-then-output algorithm structure. This allows warps to run more independently of each other, expanding multiple states until a stopping condition is reached. If the output buffer is not large enough, then the method reduces to the first solution, or worse. Also, the need to atomically access and update a single variable across many warps (the buffer "lock") can incur a performance penalty, one that is offset by reducing the amount of data sent to the CPU. For the GPU version of the BK algorithm, the size of the output buffer is the number of concurrent subtrees times a heuristic maximum clique size, determined using vertex degrees.

Load-balancing on the CPU is performed by adding *work-stealing*, requesting work from other processes at the end of Procedure 1 and re-entering the loop if work is recieved, and servicing work requests at the end of Procedure 2 if there is work to give. On the GPU, a very simple method of load balancing is performed. Each warp keeps a count of the number of nodes on its stack, stored contiguously to the output buffer. At the end of each iteration (full buffer), this list is transferred to the CPU with the buffer, sorted, and then pairs of blocks with empty stacks and blocks with large stacks *share* work, moving the bottommost half to the block with a previously empty stack. Since the number of processes is not large (typically in the hundreds), the cost of sorting and transferring the load-balance pairs is very small compared to the algorithm (about a single percent), so benefit gained from performing the sort on the GPU would be minimal. For completeness, we expect to move this routine to the GPU in the near future.

4 Benchmarking

4.1 Input Graphs

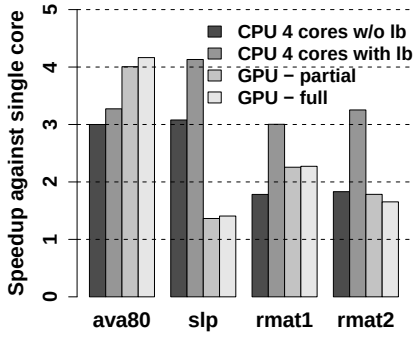
To benchmark the parallelized BK algorithm, a few graphs of varying characteristics have been chosen (see Table 2), including a functional gene-gene association network (`ava80`), climate network with Sea Level Pressure profiles between spatial grid points (`slp`) over the last 60 years, and a few synthetic graphs (`rmat-series`) generated using *GTgraph* [2], under the Recursive Matrix Graph Model (R-MAT) [4], a scale-free random graph generator.

Table 2: Input graphs.

Graph	Origin	# Vertices	# Edges	# Maximal Cliques
ava80	Biological	193,568	2,260,872	395,306
slp	Climate	10,512	679,056	365,605
rmat1	Synthetic	8,192	723,849	5,823,741
rmat2	Synthetic	32,768	3,809,695	21,903,896

4.2 GPU vs. Multi-core CPU Timing

Two differing GPU implementations are shown in Fig. 1, representing partially and fully loading a node into shared memory, compared to single-core and quad-core CPU implementations. While the GPU methods outperform the single-core CPU method in all cases, relative performance is varied against the non-load-balanced CPU version and worse than the load-balanced method. The GPU method with partial node loading performs between 1.4× and 2.25× the single-core CPU method, but up to 3× worse than the load-balanced quad-core CPU method (the speedup of ava80 is disregarded due to the very short run-time). In terms of distribution of time, the GPU transferral of cliques and load balancing accounted for between one and two percent of enumeration time, except in ava80, which was closer to ten percent. The time taken to transfer the cliques is about one percent of total time, so the buffering methodology is quite efficient compared to the enumeration process.



	ava80	slp	rmat1	rmat2
CPU 1-core	3.6	15.7	24.6	108
CPU 4-core no lb	1.2	5.1	13.8	59
CPU 4-core w/ lb	1.1	**3.8**	**8.19**	**33.2**
GPU partial	**0.9**	11.5	10.9	60.5
GPU full	**0.9**	11.2	10.8	65.3

Fig. 1: Comparison of BK algorithm between CPU and GPU. NVIDIA Tesla C2050 for GPU and Intel Xeon X5355 Quad Core 2.66GHz for CPU. Left: speedups relative to single-core performance. Right: actual time (in seconds).

5 Lessons Learned

5.1 Coarse vs. Fine-Grain Parallelization

CPU parallelized backtracking methods utilize *coarse-grain* parallelization, where multiple subtrees are explored in parallel, rather than parallelizing the work-unit itself. CPU threads/processes are *heavy-weight* in comparison to GPU threads, as they fully utilize the CPU when running and have higher context switch overhead. They also have no direct hardware dependency on other threads, as opposed to CUDA's warp-based architecture.

Coarse-grain parallelization of backtracking algorithms on the GPU is, in a naïve sense, simple to port. Call each unit executing a subtree a *process* and partition the global memory of the GPU, one "stack" for each process. To saturate the GPU hardware for a one-thread-per-process representation, a huge number of processes would be needed, leaving little to no space per process. Since the search tree and search nodes of many problems is non-uniform and cross-subtree data is non-contiguous, there would be no coalescing and high divergence, both bottlenecks to GPU performance.

For GPUs, *fine-grain* parallelization of the search nodes, or the performance of lines 2, 9, 10, and 14, is essential. In fact, the *fine-grain* implementation of the BK algorithm performs over 100× faster than the naïve *coarse-grain* method on the GPU, due to the aforementioned divergence rate and lack of coalescing. The *fine-grain* parallelization helps to prevent divergence by computing on similar work-units and enables read/write coalescence on *candidate path*s.

In terms of warp-divergence, the algorithm is reasonably efficient, occuring when *candidate path*s are small and when control code is run (such as thread zero updating a warp-level variable in shared memory). When an adjacency matrix data structure is used, the number of unique code paths for the algorithm is at most three, but for other representations (see Sec. 5.3), the diverging paths can be up to warp size. However, for the hash-table representation, this happens rarely. As a raw percentage of total branches, diverging branches occur 15–20% of the time.

A parallelization is useless if there is a poor work distribution strategy. Figure 2 shows the effect of the load-balancing strategy used in the GPU algorithm on `rmat2`. The last iterations suffer from work of too small granularity to be effectively load-balanced. Also, the `slp` graph fails to be effectively load-balanced, due to much larger cliques with relatively smaller branch factor than the other graphs (that is, much of the tree consists of linear chains), explaining the poor speedup compared to the CPU. Across all graphs, the effects of load-balancing on the GPU are not nearly as beneficial as on the CPU.

5.2 Global Memory Latency Hiding

Global memory latency on the GPU poses a challenge for performing memory-bound algorithms such as BK. Each CPU thread has a relatively large cache space to work with, helping to hide memory latency. GPUs do not have this luxury, as a large number of lightweight threads leave little thread-level caching capability, so backtracking methods on the GPU have to rely on other strategies.

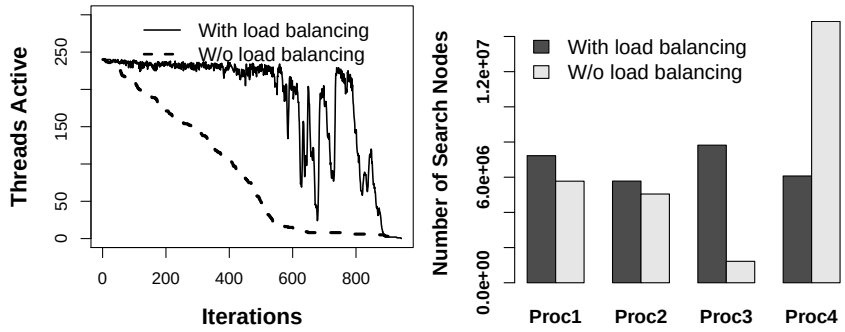


Fig. 2: The effect of load balancing over the algorithm, performed on `rmat2`. Left: GPU load-balancing. Right: CPU load balancing by process. Bars represent number of nodes expanded by process.

Latency on GPU memory operations can be hidden through a combination of coalescing, a large number of processes, and effective utilization of shared memory. Having a large number of processes, and thus a high multiprocessor occupancy, allows for some to work on the same multiprocessor while others wait for memory requests, pipelining memory operations. However, fully pipelining requires a very large number of processes, which may not be feasible. Figure 3 shows the effect of adding more concurrent subtrees. While time is decreased, the amount by which it is decreased is sub-linear, due to underutilization of hardware for small numbers of processes, non-even distribution of work, inefficiencies in load-balancing and cache contention; cache misses increased roughly proportional to the number of processes.

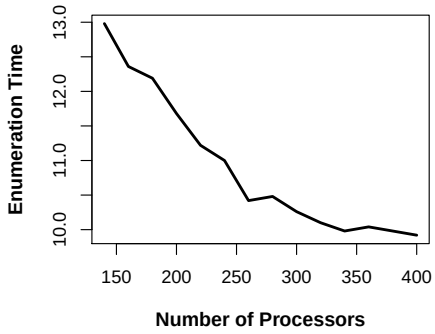


Fig. 3: The effect of adding more processes to enumeration time on the GPU.

The coalescence rate of the algorithm is approximately 20% on average, a very small number compared to optimal. One reason has to do with the global

graph data structure, see Sec. 5.3. Another is the nature of the algorithm. The average size of *candidate path* structures for the graphs tested were no more than six. This is easy to understand, as every *candidate path* representing maximal and near-maximal cliques will be small in size. In other words, opportunities for coalescence are small. This also explains the small change in performance of loading full *candidate path*s into shared memory; only a few of the search nodes actually recieve the benefit. Finally, the low coalescence rate contributes greatly to the poor performance of the algorithm relative to the CPU. Since the GPU relies on coalescing to optimize memory usage, a coalescence rate that low cannot compete with the caching capabilities of the CPU.

5.3 A Reliance on Problem Instance Representation

Backtracking algorithms have control over itermediate structures and how they are used, such as the *candidate path*, but unfortunately, there is little that can be done in a problem-independent manner to optimize global problem instances with irregular access patterns, such as graphs. This reliance on a sub-optimal problem structure is a major impediment to GPU-based algorithms, where the penalty of accessing memory without locality is much higher than on the CPU.

To minimize the penalty for accessing such structures, optimizations such as variable-packing and wide reads (such as 16 byte vs. 4 byte) can help reduce the number of these memory operations by packing data for use in register memory, such as loading multiple vertices from a graph's adjacency list. Also, utilizing the texture and constant memory of the GPU, both of which are cached, can lead to performance improvements, though the amount of each type of memory is limited and thus cannot be used for large problem instances.

For the BK algorithm, three graph representations are used, depending on memory requirements, to minimize the number of memory operations. An adjacency list and adjacency matrix are used for large and small graphs, respectively. For graphs of sizes too large for an adjacency matrix, a hash-table of neighboring vertices is used, using a simple bitwise operation between the vertex label and the table size. With the hash table, often a single index into the hash table is required to determine connectivity of two vertices, being both memory and size efficient and reducing divergence. Vertices with small degree (< 10) use a list rather than a hash table to reduce the memory footprint. To increase the chances of coalescence, parallel connectivity queries by a warp always have one vertex in common, so a similar area of the data is being accessed.

In the BK algorithm, the number of accesses to the graph are directly proportional to the number of memory operations performed on search nodes, so even with a perfect, coalesced, non-diverging algorithm on the search nodes, about 50% of the memory operations are still uncoalesced and can cause divergence, which is a large bottleneck to GPU performance. A small experiment run testing random graph connectivity queries reported a 12% coalescence rate, smaller than that in Section 5.2. It is expected that the memory inefficiency of the graph data structures is a primary cause for poor performance relative to the CPU's higher tolerance of differing access patterns.

5.4 Generality of Backtracking Properties with Respect to GPU-based Algorithms

Given a backtracking problem, the properties listed and the other lessons learned from the study of the BK algorithm can bring about meaningful insights on the feasibility of parallelizing the problem on the GPU. Having properties such as a problem instance supporting locality of access, a more regular work unit, or a more regular search tree would enable methods that would otherwise be infeasible to perform. Of course, these properties are specific to the algorithm and it is difficult to say whether a particular work unit can be parallelized in a fine grain manner or not, but given a "baseline," these algorithms can be effectively analyzed with respect to their ability to be performed on the GPU.

For example, k-d tree construction is an application in the same class as ours; Zhou et al.'s GPU implementation has important differences from our problem that allows it to compete successfully with state-of-the-art CPU implementations [20]. First, their problem domain has sufficient space to perform the tree construction in breadth-first order, eliminating the need for a stack-based representation and allowing more parallel computations. They also stress the effect of *fine-grain* parallelism on aspects of their algorithm, which we also find important. Finally, their algorithm has more computational requirements that can help hide GPU memory latency with a large enough number of threads, unlike our algorithm which is highly memory-bound. Given these properties, their algorithm competes quite well with CPU-based implementations.

6 Conclusions / Future Work

An attempt at parallelizing the backtracking paradigm, presuming the worst-case attributes against GPU performance, was presented. This problem inspires a number of future directions, despite the inability to provide good performance of MCE against a CPU. Like the k-d tree traversal algorithm on the GPU, parallelizing depth-first algorithms that do not follow the worst-case characteristics highlighted is a promising research question, one that can, under the right representation, hope to compete with or even beat their CPU-based implementations. Furthermore, other computational motifs have yet to be examined for a massively parallel machine such as a GPU. Also, as demand for general-purpose computing support on current and next-generation GPU architectures continues to grow, some of the bottlenecks (such as memory latency) may be sufficiently dealt with, leading to algorithms that could not otherwise be effectively performed on the GPU. Of course, CPU architectures continue to grow to support more throughput and parallelism, while pushing cache sizes. In either case, evaluating new and well-worn computational paradigms on state-of-the-art hardware architectures is a constant need for those who rely on them.

Acknowledgements We would like to thank Dr. W. Hendrix for useful discussions. Experiments were conducted in part on the ARC cluster support in part by NSF-CRI 0958311 and NVIDIA donations. This work was supported in part by the U.S. Department of Energy, Office of Science (SciDAC SDM Center and SciDAC Institute for Ultrascale Visualization), DOE DE-SC0005340, DOE DE-FG02-08ER25848, DE-

FC02-10ER26002/DE-SC0004935, NSF CCF-1029166, CCF-1017399, IIS-0905205, and CCF-0938000. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, 1994.
2. D. A. Bader and K. Madduri. GTgraph: A suite of synthetic random graph generators. URL: `https://sdm.lbl.gov/~kamesh/software/GTgraph/`.
3. C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.
4. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, pages 442–446, 2004.
5. T. Foley and J. Sugerman. KD-Tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005*, pages 15–22, July 2005.
6. K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. of the 2001 IEEE International Conference on Data Mining*, pages 163–170, 2001.
7. H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of Molecular Biology*, 229(3):707–721, 1993.
8. P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing*, 4873:197–208, 2007.
9. V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, 2001.
10. D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *Proc. of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 167–174, 2007.
11. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.
12. V. W. Lee, C. Kim, et al. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *Int'l Symposium on Computer Architecture*, pages 451–460, 2010.
13. J. Moon and W. Moser. On cliques in graphs. *Israel J. of Math.*, 3:23–28, 1965.
14. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
15. R. Rowe, G. Creamer, S. Hershkop, and S. J. Stolfo. Automated social hierarchy detection through email network analysis. In *9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, 2007.
16. M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *JPDC*, 69(4):417–428, 2009.
17. D. L. Tabb, M. R. Thompson, G. Khalsa-Moyers, N. C. VerBerkmoes, and W. H. McDonald. Ms2grouper: group assessment and synthetic replacement of duplicate proteomic tandem mass spectra. *Journal of the American Society for Mass Spectrometry*, 16(8):1250–61, 2005.
18. R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of GPU acceleration. *Hot Topics in Paralellism*, 35(5), 2010.
19. B. Zhang, B.-H. Park, T. Karpinets, and N. F. Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics*, 24(7):979–986, 2008.
20. K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, Dec. 2008.