2015 IEEE 21st International Conference on Parallel and Distributed Systems

Fast Execution of Simultaneous Breadth-First Searches on Sparse Graphs

Adam McLaughlin
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Adam27X@gatech.edu

David A. Bader
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
bader@cc.gatech.edu

***Abstract*—The construction of efficient parallel graph algorithms is important for quickly solving problems in areas such as urban planning, social network analysis, and hardware verification. Existing GPU implementations of graph algorithms tend to be monolithic and thus contributions from the literature are typically rebuilt rather than reused. Recent work has focused on traversal-based abstractions that efficiently execute a single breadth-first search or enact algorithms in the "think like a vertex" paradigm. However, graph analytics such as the all-pairs shortest paths problem, diameter computations, betweenness centrality, and reachability querying require the execution of many such graph traversals. Typically, these traversals are independent of one another and can thus be executed in parallel.**

This paper presents multi-search, a simple abstraction that is designed for graph algorithms requiring many breadth-first searches that can be executed simultaneously. Although algorithms have implicitly leveraged this abstraction in the past, we provide an explicit, reusable implementation that efficiently maps this abstraction to the GPU, performing more than twice as fast as previous approaches on large graphs of varying diameter. This approach allows us to scale our APSP implementation to graphs with millions of vertices using a single GPU whereas prior approaches were either constrained to much smaller graph instances or required large supercomputers to process graphs of similar size. To show the flexibility of our abstraction, we use it to express betweenness centrality and achieve more than a 5.82x average speedup over parallel CPU implementations from existing frameworks and a 2.24x average speedup over a manual, highly optimized GPU implementation of the algorithm.

I. Introduction

Graph analysis is a useful abstraction that can be applied to a variety of problems including epidemiology [28], hardware verification [19], and the modeling of physical phenomena [6]. Real world graph analytics require both scalability to large graph instances and high-performance implementations. Plenty of individual graph algorithms have been successfully accelerated using GPUs [9], [32], [33]; however, these manual implementations tend to make code reuse difficult compared to their CPU counterparts. Code reuse is tremendously important, because its absence results in tremendous effort spent duplicating and extracting work that has already been completed. Attempts to solve this issue have presented a number of abstractions that fit certain classes of graph algorithms. The Gather-Apply-Scatter (GAS) abstraction, for example, uses a generalized addition operator to pull in and accumulate information from neighboring vertices (gather), update active vertices (apply), and propagate updates to neighboring vertices (scatter) [17]. Traversal-based abstractions tend to hide the performance-sensitive details of graph traversal while requiring the user to provide application-specific functions for visiting vertices, edges, or sets of vertices or edges [34], [39], [42]. There has also been a significant effort to standardize classical graph algorithms in the context of linear algebra [24]. These abstractions tend to focus on the efficient execution of a single breadth-first search, so for problems requiring many such searches, these approaches miss out on opportunities for coarse-grained parallelism and thus, additional performance gains.

In this study we focus on graph algorithms requiring many breadth-first searches that can be executed independently in parallel. This focus isn't contrived; many analytics require searching from several (or even all) vertices in a graph for the purpose of path-counting or analyzing a graph from multiple perspectives. For example, querying which sets of vertices can reach one another can be implemented through a series of breadth-first searches, one from each vertex in the set. The All-Pairs Shortest Path (APSP) problem finds the length of the shortest path between every pair of vertices, which has been used to trace routes in transportation networks [36]. Betweenness centrality, a popular analytic for determining the most influential vertices in a graph, builds upon the APSP problem and thus also fits into this paradigm. The diameter of a graph as well as other useful graph metadata can also be determined from the solution of these problems.

We present the *multi-search* abstraction, which is a simple methodology for formulating algorithms that execute many simultaneous breadth-first searches. By providing a small number of typically short functions, the user can define his or her own algorithms that leverage this abstraction and our efficient implementation. We consider the multi-search abstraction to be a complement rather than a replacement for GAS and traversal-based paradigms. Although existing paradigms can be used to implement algorithms fitting the multi-search abstraction, we show that taking advantage of coarse-grained parallelism offered by performing many BFSs at once leads to better performance.

Based on the above, this paper presents the following contributions:

- We present the *multi-search* abstraction, a simple, yet efficient methodology for expressing algorithms that execute many graph traversals.

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.10

- We provide an efficient, cooperative implementation of this abstraction, and show that it outperforms existing implicit GPU methods by greater than a factor of 2 for large graphs of varying diameter.
- Using our abstraction, we show that a single GPU can be used to solve the APSP problem on sparse graphs with millions of vertices whereas prior art required large distributed systems to scale to graphs of similar size.
- We implement betweenness centrality using our abstraction and show more than a 5.82x average speedup over existing parallel CPU frameworks, a 3.07x average speedup over an existing GPU framework, and a 2.24x average speedup over a manual, heavily optimized GPU implementation for a diverse set of graphs.

II. Background

A. Terminology

A graph $G = (V, E)$ is an abstract representation of data consisting of a set of vertices V and a set of edges E connecting pairs of vertices. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in the graph. The work in this paper focuses on graphs with uniform edge weight, but can be extended to handle graphs with arbitrary edge weights. For simplicity, we consider graphs with undirected edges here, noting that our implementation can handle graphs with either directed or undirected edges. An undirected edge between vertices u and v can be represented as two directed edges, one from u to v and the other from v to u.

A *path* p from a vertex u to a vertex v is a set of edges starting at u and ending at v. The *degree* of a vertex u is the number of edges incident to u or the number of vertices neighboring u. The *diameter* of a graph is the length of the longest shortest path between any two vertices. The vertices of a *scale-free* graph exhibit a power-law degree distribution such that a small number of vertices have a large number of neighbors and a large number of vertices have a small number of neighbors [2]. Graphs exhibiting the *small world* phenomenon (also known as six degrees of separation) have a diameter that is logarithmic in the number of vertices [43]. Such graphs are often, but not necessarily, scale-free. Finally, a *vertex frontier* is a subset of vertices that are currently active during an iteration of a graph traversal and an *edge frontier* is the set of outgoing edges from the current vertex frontier.

B. The Multi-Search Abstraction

At a high-level, the multi-search abstraction fits any problem that requires multiple, independent breadth-first searches. We consider it a generalization of traversal-based approaches, which abstract the details of graph traversal from the operations that need to be performed on vertex frontiers. This abstraction allows domain experts in parallel algorithm design to construct the performance sensitive code that handles graph traversals and allows end users to write a small number of functions that are applied to the active vertices at each level. The end user may still need to be aware of some details of parallel programming, such as when atomic operations are necessary to avoid race conditions, but their code is typically concise and not substantially performance sensitive.

The users of our abstraction declare the set of vertices in the graph that traversals are to be executed from in addition to defining the functions that would be used in a standard traversal-based abstraction. Hence, the multi-search abstraction leverages coarse-grained parallelism because each search can be executed independently as well as fine-grained parallelism because the searches themselves can also be parallelized. In the context of GPU computing, this method of abstraction is especially useful for graphs that are too sparse to fully occupy the GPU with a single breadth-first search.

C. Multi-Search Algorithms

This subsection describes several fundamental graph algorithms that can be built on top of the multi-search abstraction. Many of these algorithms are used as subroutines themselves, showing the variety of use cases for the abstraction.

1) All-Pairs Shortest Paths: The All-Pairs Shortest Paths (APSP) problem finds the shortest paths between all pairs of vertices in a graph. The results can be represented as either the specific distances between each pair of vertices or as the paths themselves. For the latter representation, each vertex can store its parent in the breadth-first search tree that starts from the source. Note however, that the latter representation is nondeterministic as multiple valid parents may exist for a given vertex. For our experiments the choice of representation has a negligible impact on performance, so we choose to compute distances as has been done in other work in this area [7], [11], [29].

Algorithm 1: The Floyd-Warshall Algorithm

```
1 for k ← 0 ... n − 1 do
2     for i ← 0 ... n − 1 do
3         for j ← 0 ... n − 1 do
4             d[i][j] ← min(d[i][j], d[i][k] + d[k][j])
```

A canonical approach to solving the APSP problem is to use the Floyd-Warshall (FW) algorithm, shown in Algorithm 1. Algorithm 1 assumes that $d[u][v]$ is initialized to 0 when $u = v$ and the weight of the edge $u \rightarrow v$ otherwise (or ∞ if no such edge exists). Having an $O(n^3)$ complexity that is independent of the number of edges in the graph makes this approach well-suited to dense graphs. In this paper we instead focus our attention on sparse graphs, as graphs found in real world applications tend to be sparse [8], [10].

Algorithm 2: Simplified Version of Johnson's Algorithm for Sparse Graphs

```
1 for s ∈ V do
2     Q.enqueue(s)
3     while ¬Q.empty() do
4         v ← Q.dequeue()
5         for w ∈ succ(v) do
6             if d[s][w] = ∞ then
7                 Q.enqueue(w)
8                 d[s][w] ← d[s][v] + 1
```

For sparse graphs, a number of alternative approaches exist, such as Johnson's algorithm [22] or techniques based on Sparse Matrix-Vector Multiplication (SpMV) [25]. Johnson's algorithm repeatedly runs Dijkstra's Single-Source Shortest Paths (SSSP) algorithm, using every vertex in the graph as a source. For the graphs of uniform weight that we consider in this paper, this algorithm can be simplified to have $O(mn)$ complexity [5]. Algorithm 2 shows a simplified version of Johnson's algorithm for graphs with uniform weight. The algorithm assumes that initially, for each source vertex s, $d[s][t] = 0$ when $s = t$ and that $d[s][t] = \infty$ otherwise.

Since shortest path calculations are independent from one source vertex to another, implementations choose to use a *chunk*, or number of source vertices, to compute at the same time. Although the selection of a chunk size may or may not impact performance, it can have a tremendous impact on memory consumption. At one extreme, Floyd-Warshall approaches use a chunk of size n, as they compute shortest paths from all source vertices simultaneously, requiring $O(n^2)$ storage. At the other extreme, SpMV or BFS approaches uses a chunk of size 1, as they sequentially compute the shortest paths from one source vertex at a time, reusing $O(n)$ storage for the distances currently being computed (though the complete output still requires $O(n^2)$ space, of course). Our implementation is between these two endpoints: we use a chunk size equivalent to the number of streaming multiprocessors on the GPU, which is typically less than 16 on contemporary hardware. We can thus trivially scale our implementation to distributed systems with multiple GPUs per node by increasing the chunk size to be the total number of streaming multiprocessors across all GPUs on all nodes, as shown in [30] for betweenness centrality.

2) Diameter Computation: Computing the diameter of a graph once one has performed an APSP computation is trivial. The diameter can be defined in terms of d as follows:

$$diameter = \max_{u,v}\{d[u][v]\} \tag{1}$$

Although disconnected graphs have an infinite diameter, it is more helpful in practice to use a related metric, such as the diameter of the largest connected component or the effective diameter of some subset of the graph [26]. Knowledge of a graph's diameter is useful for designing the topology of computer networks in order to minimize latency, cost, and energy of sending messages between nodes [3]. Graph diameter has also been shown to have a significant performance impact for certain classes of parallel algorithms [30].

3) Transitive Closure: The *transitive closure* of a graph is the set of all reachable vertices from each vertex in the graph. Obtaining the transitive closure from d is also quite simple: if $d[u][v] \neq \infty$ then v is reachable from u, otherwise it is not. A number of trade-offs exist for computing the transitive closure of a graph. Depending on the application and particular graph being analyzed, one may prefer to store the entire transitive closure as a matrix, using $O(n^2)$ space, but providing reachability queries in $O(1)$ time. For large graphs, one may instead prefer to perform a Breadth-First Search (BFS) from the source of the query in an attempt to find the destination of the query. This approach requires just $O(1)$ space for the result but takes $O(m+n)$ time for each query. Recent research has even proposed alternative methods that fall in between these two extremes [41]. Determining whether vertices can reach one another is a fundamental graph property that has been used for memory consistency verification [31], social network analysis [12], and the LU factorization of sparse matrices [15].

4) Betweenness Centrality: An example of an algorithm that requires more work per vertex yet still fits well into the multi-search abstraction is Betweenness Centrality (BC). Betweenness Centrality is a metric that attempts to determine the most influential or important vertices (or edges) in a network. The metric quantitatively measures importance by comparing the number of shortest paths passing through a particular vertex to the total number of shortest paths found in the graph. If we let $\sigma[s][t]$ be the number of shortest paths from a vertex s to another vertex t and $\sigma[s][t](v)$ be the number of those paths that pass through a third vertex v, we can define the BC score of v as follows:

$$BC[v] = \sum_{s \neq t \neq v} \frac{\sigma[s][t](v)}{\sigma[s][t]} \tag{2}$$

Brandes defined the *dependency*, which relates the BC scores of a vertex to its successors (from the perspective of a given source vertex s) [5]:

$$\delta[s][v] = \sum_{w \in succ(v)} \frac{\sigma[s][v]}{\sigma[s][w]}(1 + \delta[s][w]) \tag{3}$$

This recursive relationship allows for the computation of betweenness centrality in two steps of the multi-search abstraction. The first is a downward traversal from s that solves the APSP problem and counts the number of shortest paths between all pairs of vertices. The second uses this information to sum dependencies between each pair of vertices back up the BFS tree until s is reached. The BC scores can be redefined in terms of the dependencies as follows:

$$BC[v] = \sum_{s \neq v} \delta[s][v] \tag{4}$$

With applications in electronic design automation, urban planning, and the analysis of the human brain, betweenness centrality has received much attention in recent literature [6], [21], [37]. BC has also been used a building block for more complicated algorithms, such as community detection [16].

III. Related Work

A. All-Pairs Shortest Paths

Noting that the APSP problem is a precursor to many other graph algorithms, it is not surprising that it has received significant attention in the literature. Prior implementations of the APSP problem tend to focus on dense graphs, distributed memory systems, and graphs containing fewer than 100,000 vertices. The APSP problem has been implemented on a rather diverse set of architectures, including FPGAs [4], GPUs [23], heterogeneous systems [29], and supercomputers [40]. We focus our work on GPU implementations of the APSP problem for large, sparse graphs representative of unstructured data found in real world applications [8].

Bondhugula *et al.* present an FPGA-based APSP implementation based on motivation from an application in bioinformatics

[4]. They developed a tiled approach to solving the Floyd-Warshall algorithm, so their methods are most effective when applied to dense graphs. Katz and Kider implement the APSP algorithm on the NVIDIA G80 architecture, also using a tiled version of FW [23]. They present results for graphs with up to $n = 11,264$ vertices and show a method for handling graphs that are larger than the amount of memory provided by a single GPU. Buluç *et al.* use a blocked recursive elimination strategy to solve the APSP problem on the GPU by noting that APSP corresponds to finding the matrix closure of the graph's adjacency matrix on the tropical semiring [7]. Using an NVIDIA GeForce 880 Ultra GPU, the authors provide an implementation that runs more than two orders of magnitude faster than an Opteron CPU. Matsumoto *et al.* provide yet another approach to computing the FW algorithm in a blocked fashion, this time on a heterogeneous CPU-GPU system [29]. Their results scale to graphs as large as $n = 43,776$ vertices, exceeding 1 TFLOPS in single-precision performance. Solomonik *et al.* implement a communication-avoiding block-cyclic APSP algorithm on a Cray XE6 supercomputer [40]. Djidjev *et al.* partition the input graph, solving the APSP problem independently on each component and unifying the results in a post-processing stage [11]. Although they focus on planar graphs, the authors present results on graphs with millions of vertices on a cluster with hundreds of GPUs.

The collection of work above focuses on algorithms that require $O(n^2)$ intermediate storage space (i.e., they use a chunk of size n). This choice of chunk size causes scalability issues on shared memory systems, requiring the deployment of these algorithms on large distributed systems to analyze graphs similar in size to the ones we study in this paper. Okuyama *et al.* instead use an approach similar to that of Johnson's algorithm and found that the cost of such an approach was that the edge distribution plays a fundamental role in the performance of the algorithm [35]. However, the presented results are shown for graphs with up to only $n = 32,768$ vertices. Our approach contributes to this area by scaling to much larger graphs with only a single GPU through the use of a smaller chunk size, achieving performance comparable to that of previous work on large distributed systems. For instance, Solomonik *et al.* solve the APSP problem for a graph with 65,536 vertices in roughly two minutes on a system with 1,024 nodes (24,576 cores) [40]. They neglect to mention the density or structure of this graph, but since their implementation is matrix-based, their performance should be roughly equivalent regardless of the graph's sparsity. For a randomly generated Delaunay mesh and Kronecker graph of the same size, our implementation requires just 41 and 99 seconds, respectively, on a single GPU. Hence, we provide a cost-effective, scalable, and fast solution to the APSP problem for sparse graphs. Furthermore, since we design our implementation as a general abstraction, other problems can leverage its results.

B. Parallel Abstractions for Graph Analysis

Recently, a number of shared memory and distributed graph programming frameworks have been developed in order to abstract the complicated details of conducting high-performance graph algorithms on parallel architectures [14], [24], [34], [39], [42]. Many of these frameworks were inspired in part by the Parallel Boost Graph Library [18]. These frameworks tend to employ abstractions in the context of linear algebra, graph traversal, or the Gather-Apply-Scatter (GAS) paradigm. Popularized by the GraphLab framework [17], the Gather-Apply-Scatter (GAS) abstraction executes algorithms by repeatedly applying three steps:

1) Gather: Collect information about vertices and edges that are adjacent to the active frontier.
2) Apply: Update the vertices in the active frontier based on the gathered information.
3) Scatter: Use these updates to determine the vertices and edges that belong to the next frontier.

Alternatively, traversal-based abstractions have the algorithm developer provide code that is applied to vertices or edges in the current frontier and sets up the frontier for the following search iteration. Finally, linear algebraic abstractions formulate graph algorithms as operations on vectors and matrices. For instance, a breadth-first search can be represented as a SpMV between the adjacency matrix of the graph and a vector representing the active vertex frontier on the (min, +), or tropical, semiring. The GraphBLAS standardizes a common set of building blocks for graphs through the language of linear algebra [24]. Our abstraction differs from a sparse matrix product in that the user writes a function to visit vertices instead of defining a semiring, and is perhaps more general because of this difference. We also consider it more intuitive for the user to write a function in terms of vertex frontiers than to define a semiring.

IV. Multi-Search Implementation

Our work complements existing paradigms as it provides a related abstraction that focuses on problems that require many graph traversals that can be executed independently. We view the multi-search abstraction as a generalization of the traversal-based abstraction: the more coarse-grained parallelism that is available, the more likely one will benefit through the use of the multi-search paradigm rather than the traversal-based paradigm. In the event that only one search is desired, the multi-search abstraction simply reduces to the traversal-based abstraction. This section presents an efficient, cooperative implementation of the multi-search abstraction that can be used to develop a number of useful graph algorithms (such as the ones described in Section II-C). Similar to GAS and traversal-based methods, the multi-search abstraction derives its utility from decoupling the complicated details of the underlying graph traversals from the specific updates that need to occur for the higher-level algorithm at hand. Hence, users are only required to implement a few small functions that can all utilize the same device kernel for graph traversals, encouraging code reuse and alleviating programmers from having to implement their own error-prone sets of parallel graph traversals.

Algorithm 3 shows our cooperative implementation of the multi-search abstraction. Careful implementation of the abstraction is rather important, since it will profoundly affect the performance of all the algorithms that leverage the abstraction. Users of the abstraction implement a small number of functions:

- `init()`: Initialize data structures at the beginning of a search from source i.
- `prior()`: Handle any computation that may occur just prior to a search iteration.

Algorithm 3: Pseudocode for the Multi-Search Abstraction Kernel

```
   // Loop across SMs
1  for i ∈ S do in parallel
2      Q_curr[0] ← i
3      Q_curr_len ← 1
4      Q_next_len ← 0
5      init(i)
6      barrier()
7      while Q_curr_len ≠ 0 do
8          prior()
9          barrier()
10         for v ∈ Q_curr do
               // Loop across threads
11             for w ∈ neighbors(v) do in parallel
12                 visitVertex(i, v, w)
13         swap(Q_curr, Q_next)
14         barrier()
15         Q_curr_len ← Q_next_len
16         Q_next_len ← 0
17         post()
18         barrier()
19     finalize(i)
```

- `visitVertex()`: When an edge (u, v) is traversed from source i, update the appropriate data structures in terms of u, v, and i.
- `post()`: Handle any computation that may occur at the end of a search iteration.
- `finalize()`: Handle any computation that may occur after the search from source i is complete.

These functions are made available for generality and convenience to the end user; it isn't expected that the user will necessarily need all of these functions for their applications. Algorithm 3 represents just one implementation of the abstraction for a set of hardware problems. Others are possible, and users of the abstraction won't need to change their code when implementations of the abstraction are improved.

The `for` loop on Line 1 is executed in parallel across the Streaming Multiprocessors (SMs) of the GPU. In practice, the user's case may only require a subset of vertices to search, so we define the variable S to be this user-defined set. For the APSP problem, $S = V$. The number of vertices in the graph vastly outnumbers the number of SMs on the GPU, so each SM sequentially processes many iterations of this for loop. The while loop on Line 7 is executed by all of the threads within each SM. We organize work at the warp level, where a warp is a group of (as of this writing) 32 threads that execute in lockstep within each SM. Initially we assigned each thread to its own queue element and had warps cooperatively process the adjacency lists of these elements one at a time. Although this approach sufficiently balanced the work among threads within each warp, it left an imbalance of work *between* warps. For sufficiently small queues one warp could be left processing the entire frontier while the other warps idle. Improving upon this approach, we implemented a dynamic scheduling policy that has warps asynchronously dequeue vertices in the current vertex frontier. Instead of being statically assigned explicit batches of vertices within each vertex frontier to process, warps dequeue the next vertex after processing their current vertex. Hence, a warp only will idle when there is no work remaining in the current frontier.

The threads within each warp cooperatively process the edges outgoing from to the dequeued vertex collected by that warp. This cooperation leverages the `__shfl()` intrinsic introduced by NVIDIA's Kepler architecture. The shuffle intrinsic allows for fast communication within a warp without requiring the use of shared memory. For instance, `__shfl(x,y)` returns the value of `x` held by thread `y` to all of the other threads in the warp. Each thread in the warp traverses consecutive outgoing edges from that queue element.

Algorithm 4: Implementation of `init()` for the All-Pairs Shortest Paths Problem

```
1 for k ∈ |V| do in parallel
2     if k = i then
3         d[i][k] ← 0
4     else
5         d[i][k] ← ∞
```

Algorithm 5: Implementation of `visitVertex()` for the All-Pairs Shortest Paths Problem

```
1 if d[i][w] = ∞ then
2     d[i][w] ← d[i][v] + 1
3     t ← atomicAdd(&Q_next_len, 1)
4     Q_next[t] ← w
```

Implementing the user-defined functions to implement the APSP algorithm on top of the multi-search abstraction is fairly straightforward. Algorithms 4 and 5 show APSP-specific implementations of `init()` and `visitVertex()`, respectively. The implementations for `prior()`, `post()`, and `finalize()` can be left empty for this algorithm but may be necessary for more complicated algorithms. Algorithm 4 simply initializes $d[u][v]\ \forall\ u, v \in V \times V$. Algorithm 5 simply checks if w has been visited. If not, it is atomically added to Q_{next} to avoid race conditions. Note that duplicate entries in the queue are possible, since multiple threads may see w as unvisited before the first of these threads writes to $d[i][w]$. In practice, these duplicates are rare because they require either duplicate edges or multiple warps to simultaneously execute the same instruction. In our tests we found that the `atomicAdd()` on Line 3 was faster than having each warp prefix scan whether or not it found an unvisited vertex. This result makes sense because the atomic operation is with respect to a location in shared memory and because warps would have to perform their own scans since each warp expands a different queue element. For algorithms that require the *number* of shortest paths between each pair of vertices, an atomic Compare and Swap (CAS) operation must be used to set the distances of unvisited vertices. Otherwise, certain paths could be double counted, leading to incorrect results. Although atomic operations are required here, in the future we plan to provide our own queue data structure

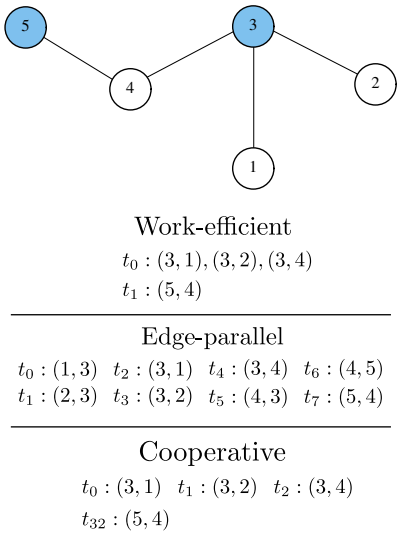


Fig. 1. Several thread decompositions for the multi-search abstraction

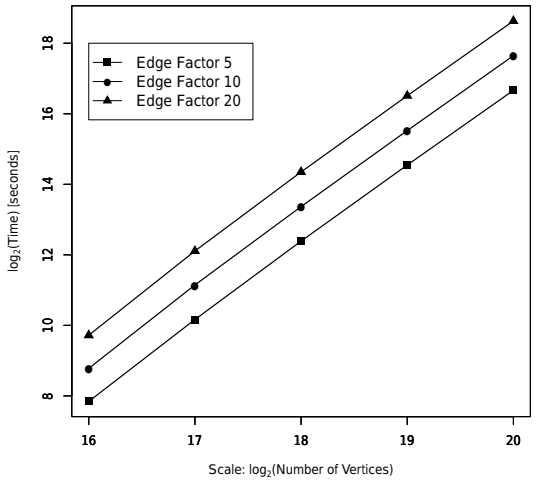


Fig. 2. Impact of scaling vertices and edges on performance for Erdős-Rényi graphs

that provides a "safe" enqueuing function to abstract such operations away from end users.

Several other methods for solving problems fitting this paradigm on the GPU are essentially algorithms that solve the APSP problem without the explicit use of this abstraction. For instance, Jia et al. present vertex and edge-parallel methods for computing betweenness centrality, noting that the edge-parallel approach better maximizes the memory bandwidth achieved by the GPU [20]. Similarly, McLaughlin and Bader provide an approach that works particularly well for computing the betweenness centrality of high-diameter graphs [30]. Both of these methods employ an approach to BC that reflects the multi-search abstraction in that the APSP problem is solved for each vertex before dependencies are computed. Hence, we can compare their mappings of simultaneous graph traversals to the GPU.

Figure 1 shows an example of how these methods differ for a simple graph. Consider a streaming multiprocessor that has been assigned a breadth-first search from vertex 4. In the first iteration of this search, vertex 4 will enqueue vertices 3 and 5, which become the active vertex frontier for the next iteration of the search. Figure 1 reflects this state, as vertices 3 and 5 are marked as active (shaded). Beneath the picture of the graph in Figure 1 we show how the work-efficient approach [30], the edge-parallel approach [20], and our cooperative approach from Algorithm 3 assign the threads within the SM to edge traversals. The work-efficient approach assigns threads within each SM of the GPU to vertices on the active frontier. Hence, the first thread traverses the three outgoing edges from vertex 3 and a second thread traverses the outgoing edge from vertex 5. Note that although this method only traverses edges in the active edge frontier, threads have data-dependent amounts of work to do and hence the amount of work per thread can vary tremendously using this approach, leading to potentially severe load imbalances. The edge-parallel approach simply assigns a thread to every edge in the graph, regardless of whether or not it is in the active frontier. This approach easily occupies the GPU as many threads are needed to process iterations of moderate size; however, not all of these threads are contributing to the progress of the algorithm. Finally, our cooperative approach assigns warps to the adjacency lists of each vertex in the active frontier one by one. Hence, the three outgoing edges of vertex 3 are processed by 3 threads of the one warp and the lone edge from vertex 5 is then processed by one thread from a second warp, all in parallel. When graphs are sufficiently large such that the average adjacency list of a vertex tends to be greater than the warp size of the architecture, the utilization of each warp is high and the only edges processed by threads within each warp are edges in the current edge frontier. Furthermore, since this process is cooperative, threads have a well-partitioned amount of work.

Figure 2 shows the scalability of our cooperative approach presented in Algorithm 3 when it is used to solve the All-Pairs Shortest Paths problem. We use randomly generated Erdős-Rényi graphs and vary the number of vertices and edges to see how these changes impact performance. The edge factor influences the probability p of an edge selection in the $G(n,p)$ Erdős-Rényi random model [13]. Intuitively, a graph generated with twice the edge-factor will contain twice as many edges; however, the edge-factor should not be mistaken for average degree. The average degree of graphs with edge-factor 5 used to generate Figure 2 is approximately 28.

Figure 2 shows that our methodology is robust to scaling both the number of vertices and the number of edges in the graph. On average, increasing the edge factor by two results in a 1.96x increase in execution time. One reason for why this increase in execution time is slightly less than the expected theoretical increase of 2x is that additional edges can provide better warp occupancy. For instance, if the degree of a vertex modulo the warp size of the architecture is small (but nonzero), additional edges will only give unoccupied threads work until all threads are occupied. Increasing the number of vertices by a factor of two (which, for the same edge factor, also increases the number of edges by a factor of two) results in an increase in execution time of 4.64x on average. Since the computational complexity of the algorithm is $O(mn)$, we theoretically expect to see a factor of 4 increase in execution

TABLE I. GRAPH DATASETS USED FOR THIS STUDY. NODES AND EDGES ARE DISPLAYED IN MILLIONS.

Graph	Nodes	Edges	Notes/Sparsity	
333SP	3.71m	22.22m	Ferrari	
adapative	6.82m	27.25m	Urban Sim.	
as-Skitter	1.70m	22.19m	Internet	
auto	0.45m	6.63m	Partitioning	
delaunay_n21	2.10m	12.58m	Triangulation	
ecology1	1.00m	4.00m	Gene Flow	
hollywood-2009	1.14m	115.03m	Movie Actors	
kron_g500-logn19	0.52m	43.56m	Kronecker	
ldoor	0.95m	45.57m	Large Door	
roadNet-CA	1.96m	5.53m	Intersections	
rgg_n_2_21_s0	2.10m	28.98m	Geometric	
thermal2	1.23m	7.35m	Diffusion	

time. The additional 0.64x increase that we see in practice could result from contention in resources when accessing memory atomically as well as from potential load imbalances among SMs.

V. EXPERIMENTAL SETUP

In the next section we present performance results that show the scalability of the multi-search abstraction as well as how it performs for several classes of real-world graphs. To show the utility of the abstraction itself we implement betweenness centrality on top of it and compare the performance of our method to that of recent literature. CPU results were run on an Intel Core i7-2600K processor. The Core i7-2600K has a frequency of 3.4 GHz, and 8 MB last level cache, four physical processor cores and a peak memory bandwidth of 21 GB/s. We show results for CPU tests using 4 threads, since the use of hyperthreading didn't improve performance. GPU results were run on NVIDIA GeForce GTX Titan and NVIDIA Tesla K40c GPUs. The GeForce GTX Titan is a compute capability 3.5 GPU designed under the Kepler architecture that has 14 streaming multiprocessors, 6 GB of device memory, a clock frequency of 837 MHz, and a peak memory bandwidth of 288.4 GB/s. The Tesla K40c is another compute capability 3.5 Kepler GPU that has 15 streaming multiprocessors, 12 GB of device memory, a clock frequency of 725 MHz, and the same peak memory bandwidth as the GeForce GTX Titan.

CPU code was compiled using `g++` version 4.8.1 and OpenMP. GPU code was compiled using `nvcc` and the CUDA 7.0 toolkit, which we leverage for C++11 support in device functions, allowing us to use lambda functions to implement the user-defined portions of the multi-search abstraction[1]. We present results based on publicly available graph data sets from the 10th DIMACS Challenge [1], the Stanford Network Analysis Platform [27], and the University of Florida Sparse Matrix Collection [10]. Table I shows more information about the set of graphs we perform tests on, including the number of vertices and number of (directed) edges for each graph, the significance of each data set, and finally the sparsity pattern of each data set. Note that we use both real-world and randomly generated graphs with highly varying connectivity from regular numerical meshes to irregular scale-free graphs.

For scaling experiments, we compare against the work-efficient [30] and edge-parallel approaches [20] described in Section IV and contrasted in Figure 1. These techniques are GPU-based and all of these experiments were run on the Tesla K40c GPU. For the experiments on the benchmarks in Table I, we compare against both CPU and GPU implementations, where all GPU experiments were run on the GeForce GTX Titan GPU.

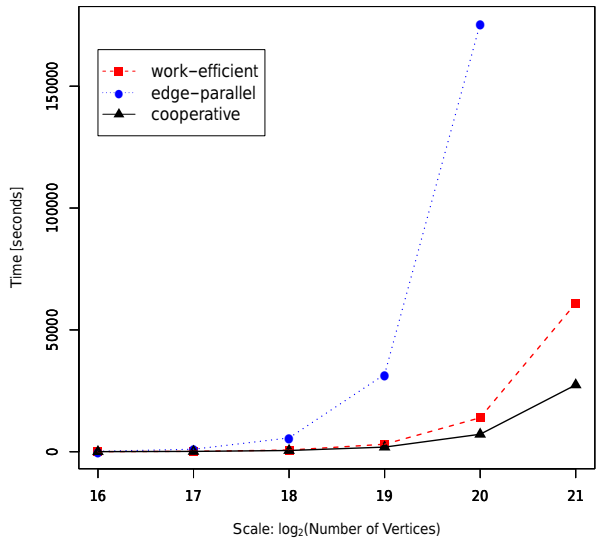
VI. EXPERIMENTAL RESULTS

A. All-Pairs Shortest Paths

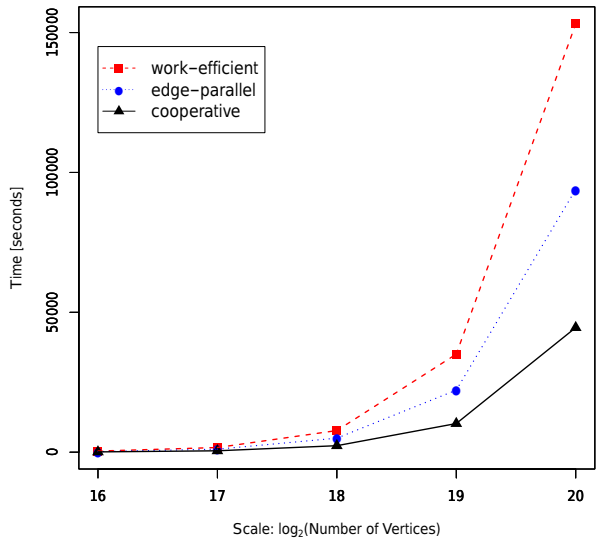
Figure 3 compares APSP execution times using the three methods of graph traversal shown in Figure 1. Figure 3a shows results for a high-diameter Delaunay mesh, which typically requires hundreds of search iterations to find all reachable vertices from a given source vertex. In contrast, Figure 3b shows results for a low-diameter, scale-free Kronecker graph, which typically requires fewer than ten search iterations to complete a Breadth-First Search. We can see that for the Delaunay mesh, the work-efficient approach from [30] is preferential to the edge-parallel approach from [20] whereas for the Kronecker graph, the opposite is true. This notion that the graph structure has significant performance implications lead to the hybrid approach presented in [30]. However, we can see that for both of these classes of graphs, our cooperative approach from Algorithm 3 is more robust to the structure of the graph, performing more than twice as fast on large scales of both high and low-diameter graphs. The work-efficient approach performs poorly on scale-free networks since the power-law distribution of vertex degree leads to severe load imbalances among threads. The edge-parallel approach, in contrast, does better on scale-free networks because a larger percentage of edges are active at once and since threads have an equivalent amount of work. However, this approach performs poorly on the smaller vertex frontiers seen in high-diameter graphs since a majority of threads will be assigned to edges that don't actually need to be inspected. The cooperative approach alleviates these issues by having the warps within each SM asynchronously process adjacency lists, allowing for work-efficiency as well as sufficient load-balancing among concurrent threads.

Table II shows the time required to solve the APSP problem as well as the maximum degree for our set of benchmark graphs. We include the maximum degree simply to enhance the information given in Table I, where we could not fit it. Using a single GPU, we can scale to significantly larger graph instances in comparison to existing methods, since our approach uses a relatively small chunk of simultaneous source vertices. Existing implementations tend to either use large distributed systems to

[1]Source code: https://github.com/Adam27X/graph-utils/



(a) Triangular mesh



(b) Scale-free network

Fig. 3. Comparison of multi-search traversal techniques for two classes of networks

TABLE II. BENCHMARK RESULTS FOR SOLVING THE APSP PROBLEM.

Graph	APSP Time (s)	Maximum Degree
333SP	93150	28
adapative	342253	4
as-Skitter	28333	35455
auto	2291	37
delaunay_n21	27432	23
ecology1	9187	4
hollywood-2009	43082	11469
kron_g500-logn19	10236	80674
ldoor	7802	76
roadNet-CA	34358	12
rgg_n_2_21_s0	49991	37
thermal2	13349	10

scale to graphs this large [11], [40] or restrict their studies to smaller instances of graphs on shared memory machines [4], [7], [23], [29].

It is intriguing to note that our performance on *ldoor* is better than that of *kron_g500-logn19* despite the fact that *ldoor* is a slightly larger graph. The irregularity of Kronecker graphs makes them particularly challenging to process. Significant warp divergences may occur when one warp processes the vertex with the largest number of edges, causing other warps in the block to idle before moving on to the next vertex frontier. Splitting vertices with especially large frontiers into virtual vertices is one potential improvement that could alleviate this issue [38] that we intend to explore for future work.

B. Betweenness Centrality

Since we present this work as an abstraction that can be applied to a number of problems requiring many simultaneous breadth-first searches, it is important to show the performance of such problems under our abstraction. We compare our approach to four implementations of Betweenness Centrality from recent literature: Ligra [39], Galois [34], Gunrock [42], and a hybrid GPU implementation from [30]. Ligra is a shared-memory CPU graph framework that uses a traversal-based abstraction that allows users to write graph algorithms that map over frontiers of edges and vertices (or subsets thereof). Galois is a CPU-based system that provides the user with parallel set iterators, allowing the user to write sequential code that specifies loops that should be run in parallel. Rather than using the bulk synchronous parallel model of execution, Galois uses worklists to implement asynchronous execution. Gunrock has a similar programming interface to Ligra, but is written in CUDA for execution on GPU backends. It includes an advance stage that visits the current vertex frontier as well as a filter stage that generates the next frontier. Ligra, Galois, and Gunrock provide their own implementations of betweenness centrality, which we use for our experiments. Finally, the hybrid_BC GPU implementation of betweenness centrality uses an on-line approach to determine whether a graph will benefit more from either the work-efficient or edge-parallel methods that were shown in Figure 1. In terms of programmability, Galois is the most general as it can implement any worklist-based algorithm, Ligra and Gunrock are specialized to traversal-based graph algorithms, and hybrid_BC is a manual implementation that is specialized solely for betweenness centrality. Our multi-search abstraction is meant for algorithms requiring many graph traversals, but could be specialized to act similarly to Gunrock or Ligra in the event that a sufficient number of traversals aren't available for the user's application.

Table III shows timing results for each of these baselines as well as our own cooperative approach. Table IV summarizes the results in Table III by showing the average speedup our cooperative approach attains over the methods that we compare to from prior literature. For all tests we approximate BC scores using $k = 8192$ source vertices to make the running time of the algorithm more reasonable. The approximation simply performs APSP calculations and dependency accumulations from k source vertices rather than all of them, so the time to compute the exact BC scores is roughly $\frac{n}{k}$ times the time to compute the approximate scores.

TABLE III. BENCHMARK RESULTS FOR COMPUTING BETWEENNESS CENTRALITY. TIMES ARE IN SECONDS. THE FASTEST RESULT FOR EACH GRAPH IS PRESENTED IN BOLD.

Framework	*333SP*	*adaptive*	*as-Skitter*	*auto*	*delaunay_n21*	*ecology1*
Galois	4651	7086	1167	637	2004	906
Ligra	3005	3442	1241	665	992	635
Gunrock	1999	4851	N/A	161	712	1458
hybrid_BC	781	993	518	407	373	176
Cooperative	**352**	**601**	**275**	**74**	**174**	**104**

Framework	*hollywood-2009*	*kron_g500-logn19*	*ldoor*	*roadNet-CA*	*rgg_n_2_21_s0*	*thermal2*
Galois	2058	1868	1240	1498	3518	1088
Ligra	4318	623	1751	700	2808	899
Gunrock	630	**406**	395	N/A	N/A	277
hybrid_BC	1591	522	621	403	1066	204
Cooperative	**602**	523	**183**	**145**	**399**	**115**

TABLE IV. AVERAGE SPEEDUP OF THE COOPERATIVE APPROACH OVER EXISTING FRAMEWORKS.

	Galois	Ligra	Gunrock	hybrid_BC
Speedup of Cooperative	**7.66x**	**5.82x**	**3.07x**	**2.24x**

Compared to the parallel CPU implementations of Galois and Ligra our implementation does very well, averaging 7.66x and 5.82x speedups, respectively. The results in comparison to Gunrock are more interesting in that they vary tremendously. Since Gunrock uses a chunk size of 1 (i.e. it only leverages fine-grained parallelism for BC), it does particularly poorly on graphs with low average degree, such as *ecology1* and *adaptive*. On the other hard, Gunrock performs well on graphs that do offer lots of fine-grained parallelism, such as *hollywood-2009*, where its performance is competitive with ours and *kron_g500-logn19* where its performance is even better than our own. The entries denoted "N/A" in Table III for Gunrock correspond to graphs that caused a memory access violation on the GPU. For the graphs that we could compare, our implementation was 3.07x faster on average than Gunrock. Finally, our GPU abstraction is competitive with GPU code that is specialized for computing BC scores. The hybrid_BC implementation is never significantly faster than that of our own yet for *ldoor* our approach is 3.41x faster. Overall, our cooperative approach is 2.24x faster than hybrid_BC on average and is much more easily leveraged for the development of algorithms that can take advantage of the multi-search abstraction. Even though hybrid_BC is specialized for BC, our approach is faster because of our efficient implementation of graph traversals shown in Algorithm 3 and Figure 1.

VII. CONCLUSIONS

In this paper, we present and provide an efficient implementation of an abstraction for processing many simultaneous breadth-first searches in parallel on the GPU. We implement the abstraction by enlisting the threads within each warp to cooperatively traverse the edges in elements in the active vertex frontier. This approach is more than twice as fast as previous GPU approaches that were used to schedule threads for simultaneous graph traversals for large graphs of both low and high diameter. Furthermore, our approach scales to graphs with millions of vertices using a single GPU whereas previous approaches used large clusters to solve problems of similar size in greater amounts of time. Finally, our abstraction can efficiently implement more complicated algorithms. We show that an implementation of betweenness centrality that leverages our abstraction achieves an average speedup of 7.66x and 5.82x over the Galois and Ligra multi-core graph frameworks, a 3.07x speedup over the Gunrock GPU graph framework, and an average speedup of 2.24x over a heavily optimized on-line GPU implementation of betweenness centrality.

The literature on parallel implementations of graph algorithms is beginning to shift from manual, hand-tuned implementations of specific algorithms to libraries that provide abstractions for certain classes of parallel algorithms. The appropriate choice of an abstraction depends on the problem that needs to be solved, the way in which each abstraction is mapped to hardware, and the graph being analyzed. We consider the definition of new abstractions and the unification existing abstractions into a general parallel graph analytics framework to be an exciting area of future work.

ACKNOWLEDGMENT

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala). Finally, we would like to thank NVIDIA Corporation for their donation of GeForce GTX Titan and Telsa K40 GPUs.

REFERENCES

[1] D. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for Graph Clustering and Partitioning," in *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj and J. Rokne, Eds. Springer New York, 2014, pp. 73–82.

[2] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[3] M. Besta and T. Hoefler, "Slim Fly: A Cost Effective Low-diameter Network Topology," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 348–359.

[4] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based All-pairs Shortest-paths in a Directed Graph," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 112–112.

[5] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[6] E. Bullmore and O. Sporns, "Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems," *Nature Reviews Neuroscience*, vol. 10, no. 3, pp. 186–198, 2009.

[7] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU," *Parallel Computing*, vol. 36, no. 5, pp. 241–253, 2010.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[9] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single Source Shortest Paths," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp. 349–359.

[10] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[11] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier, "Efficient Multi-GPU Computation of All-Pairs Shortest Paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 360–369.

[12] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive Social Network Analysis: Mining Twitter for Social Good," in *IEEE International Conference on Parallel Processing (ICPP)*, 2010, pp. 583–593.

[13] P. Erdós and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungar. Acad. Sci*, vol. 5, pp. 17–61, 1960.

[14] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 2014, pp. 1–6.

[15] J. R. Gilbert and J. W. Liu, "Elimination structures for unsymmetric sparse LU factors," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 2, pp. 334–352, 1993.

[16] M. Girvan and M. E. Newman, "Community Structure in Social and Biological Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." *OSDI*, vol. 12, no. 1, p. 2, 2012.

[18] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[19] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[20] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge v. Node Parallelism for Graph Centrality Metrics," *GPU Computing Gems*, vol. 2, pp. 15–30, 2011.

[21] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong, "A Novel Application of Parallel Betweenness Centrality to Power Grid Contingency Analysis," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–7.

[22] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977.

[23] G. J. Katz and J. T. Kider, Jr, "All-pairs Shortest-paths for Large Graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55.

[24] J. Kepner, D. A. Bader, A. Buluc, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, Matrices, and the GraphBLAS: Seven Good Reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[25] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[26] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 177–187.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[28] F. Liljeros, C. R. Edling, L. A. Amaral, H. E. Stanley, and Y. Aberg, "The Web of Human Sexual Contacts," *Nature*, vol. 411, no. 6840, pp. 907–908, 2001.

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

[30] A. McLaughlin and D. A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

[31] A. McLaughlin, D. Merrill, M. Garland, and D. A. Bader, "Parallel Methods for Verifying the Consistency of Weakly-Ordered Architectures," in *Proceedings of the 24th ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[32] A. McLaughlin, J. Riedy, and D. A. Bader, "Optimizing Energy Consumption and Parallel Performance for Static and Dynamic Betweenness Centrality using GPUs," in *Eighteenth IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[33] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[34] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 456–471.

[35] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all-pairs shortest paths using the gpu," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, 2012.

[36] S. Pallottino and M. G. Scutella, "Shortest path algorithms in transportation models: classical and innovative aspects," in *Equilibrium and advanced transportation modelling*. Springer, 1998, pp. 245–281.

[37] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora, "Street Centrality and Densities of Retail and Services in Bologna, Italy," *Environment and Planning B: Planning and design*, vol. 36, no. 3, pp. 450–465, 2009.

[38] A. E. Sariyuce, E. Saule, K. Kaya, and U. V. Catalyurek, "Regularizing Graph Centrality Computations," *Journal of Parallel and Distributed Computing*, vol. 76, no. 0, pp. 106 – 119, 2015, special Issue on Architecture and Algorithms for Irregular Applications.

[39] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[40] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 548–559.

[41] R. R. Veloso, L. Cerf, W. M. Junior, and M. J. Zaki, "Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach." in *EDBT*, 2014, pp. 511–522.

[42] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, no. 1501.05387v1, Jan. 2015.

[43] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.