

A High-Performance Connected Components Implementation for GPUs

Jayadharini Jaiganesh
Department of Computer Science
Texas State University
San Marcos, TX 78666, USA
jayadharini@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX 78666, USA
burtscher@txstate.edu

ABSTRACT

Computing connected components is an important graph algorithm that is used, for example, in medicine, image processing, and biochemistry. This paper presents a fast connected-components implementation for GPUs called ECL-CC. It builds upon the best features of prior algorithms and augments them with GPU-specific optimizations. For example, it incorporates a parallelism-friendly version of pointer jumping to speed up union-find operations and uses several compute kernels to exploit the multiple levels of hardware parallelism. The resulting CUDA code is asynchronous and lock free, employs load balancing, visits each edge exactly once, and only processes edges in one direction. It is 1.8 times faster on average than the fastest prior GPU implementation running on a Titan X and faster on most of the eighteen real-world and synthetic graphs we tested.

CCS Concepts

• **Computing methodologies** → Concurrent algorithms; Massively parallel algorithms

Keywords

connected components; graph algorithm; union-find; parallelism; GPU implementation

ACM Reference format:

J. Jaiganesh and M. Burtscher. 2018. A High-Performance Connected Components Implementation for GPUs. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing, Tempe, Arizona USA, June 2018 (HPDC'18)*, 13 pages. DOI: 10.1145/3208040.3208041

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '18, June 11–15, 2018, Tempe, AZ, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5785-2/18/06...\$15.00
<https://doi.org/10.1145/3208040.3208041>

1 INTRODUCTION

A connected component of an undirected graph $G = (V, E)$ is a maximal subset S of its vertices that are all reachable from each other. In other words, $\forall a, b \in S$ there exists a path between a and b . The subset is maximal, i.e., no vertices are reachable from it that are not in the subset. The goal of the connected-components (CC) algorithms we study in this paper is to identify all such subsets and *label* each vertex with the ID of the component to which it belongs. A non-empty graph must have at least one connected component and can have as many as n , where $n = |V|$.

For illustration purposes, assume the cities on Earth are the vertices of a graph and the roads connecting the cities are the edges. In this example, the road network of an island without bridges to it forms a connected component.

Computing connected components is a fundamental graph algorithm with applications in many fields. For instance, in medicine, it is used for cancer and tumor detection (the malignant cells are connected to each other) [16]. In computer vision, it is used for object detection (the pixels of an object are typically connected) [14]. In biochemistry, it is used for drug discovery and protein genomics studies (interacting proteins are connected in the PPI network) [39]. Fast CC implementations, in particular parallel implementations, are desirable to speed up these and other important computations.

Determining CCs serially is straightforward using the following algorithm. After marking all vertices as unvisited, visit them in any order. If a visited vertex is still marked as unvisited, use it as the source of a depth-first search (DFS) or breadth-first search (BFS) during which all reached vertices are marked as visited and their labels are set to the unique ID of the source vertex. This takes $O(n+m)$ time, where $m = |E|$, as each vertex must be visited and each edge traversed.

Alternatively, CCs can be found with the following serial algorithm. First, iterate over all vertices and place each of them in a separate disjoint set using the vertex's ID as the representative (label) of the set. Then, visit each edge (u, v) and combine the set to which vertex u belongs with the set to which vertex v belongs. This algorithm returns a collection of sets where each set is a connected component of the graph. Determining which set a vertex belongs to and combining it with another set can be done very efficiently using a disjoint-set data structure (aka a union-find data structure) [9]. Asymptotically, this algorithm is only slower

than the DFS/BFS-based algorithm by the tiny factor $\alpha(n)$, i.e., the inverse of the Ackermann function.

Whereas there are efficient parallel implementations that are based on the BFS approach, for example Ligra+ BFSCC [31], the majority of the parallel CC algorithms are based on the following “label propagation” approach. Each vertex has a label to hold the component ID to which it belongs. Initially, this label is set to the vertex ID, that is, each vertex is considered a separate component, which can trivially be done in parallel. Then the vertices are iteratively processed in parallel to determine the connected components. For instance, each vertex’s label can be updated with the smallest label among its neighbors. This process repeats until there are no more updates, at which point all vertices in a connected component have the same label. In this example, the ID of the minimum vertex in each component serves as the component ID, which guarantees uniqueness. To speed up the computation, the labels are often replaced by “parent” pointers that form a union-find data structure.

Our ECL-CC implementation described in this paper is based on this approach. However, ECL-CC combines features and optimizations in a new and unique way to yield a very fast GPU implementation. The paper makes the following main contributions.

- It presents the ECL-CC algorithm for finding and labeling the connected components of an undirected graph. Its parallel GPU implementation is faster than the fastest prior CPU and GPU codes on average and on most of the eighteen tested graphs.
- It shows that “intermediate pointer jumping”, an approach to speed up the find operation in union-find computations, is faster than other pointer-jumping versions and essential to the performance of our GPU implementation.
- It describes important code optimizations for CC computations that are necessary for good performance but that have heretofore never been combined in the way ECL-CC does.

The CUDA code is available to the research and education community at <http://cs.txstate.edu/~burtscher/research/ECL-CC/>.

The rest of the paper is organized as follows. Section 2 provides background information and discusses related work. Section 3 describes the ECL-CC implementation. Section 4 explains the evaluation methodology. Section 5 presents the experimental results. Section 6 concludes the paper with a summary.

2 BACKGROUND AND RELATED WORK

A large amount of related work exists on computing connected components. In this section, we primarily focus on the shared-memory solutions that we compare against in the result section as well as the work on which they are based. This paper does not consider *strongly* connected components as they are of interest in *directed* graphs.

Hopcroft and Tarjan’s article is one of the first to describe a serial connected-components algorithm [15]. However, they mention that the algorithm was already well known. It has linear time complexity as it visits each vertex once. Many later CC algorithms are based on this work. Various graph-processing libraries such as Boost [32], Galois [19], igraph [5], and Lemon [6] include serial code to compute connected components.

Fig. 1 illustrates the parallel label-propagation approach outlined in the introduction, where the labels are propagated through neighboring vertices until all vertices in the same component are labelled with the same ID. First, the labels (small digits) are initialized with the unique vertex IDs (large digits) as shown in panel (a). Next, each vertex’s label is updated in parallel with the lowest label among the neighboring vertices and itself (b). This step is iterated until all labels in the component have the same value (c).

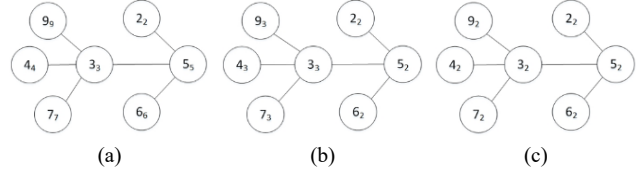


Fig. 1. Example of label propagation (large digits are unique vertex IDs, small digits are label values): (a) initial state, (b) intermediate state, (c) final state

For performance reasons, the labels are typically not propagated in this manner. Instead, they are indirectly represented by a parent pointer in each vertex that, together, form a union-find data structure. Starting from any vertex, we can follow the parent pointers until we reach a vertex that points to itself. This final vertex is the “representative”. Every vertex’s implicit label is the ID of its representative and, as before, all vertices with the same label (aka representative) belong to the same component. Thus, making the representative u point to v indirectly changes the labels of all vertices whose representative used to be u . This makes *union* operations very fast. A chain of parent pointers leading to a representative is called a “path”. To also make the traversals of these paths, i.e., the *find* operations, very fast, the paths are sporadically shortened by making earlier elements skip over some elements and point directly to later elements. This process is referred to as “path compression”.

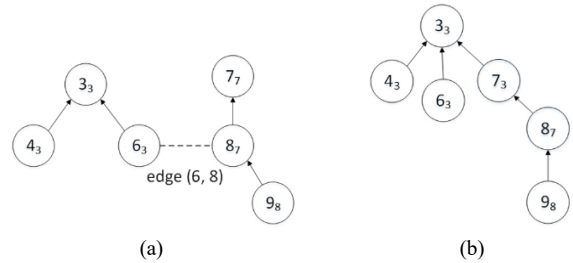


Fig. 2. Example of hooking operation: (a) before hooking, (b) after hooking

Shiloach and Vishkin proposed one of the first parallel CC algorithms [28]. It is based on the union-find approach and employs two key operations called “hooking” and “pointer jumping”. Hooking, which is the union operation, works on edges and exploits the fact that the vertices on either end of an edge must belong to the same component. For a given edge (u, v) , the hooking operation checks if both vertices have the same parent. If not, the parents’ vertex IDs are compared. If the parent with the higher ID is a representative, it is made to point to the other parent with the

lower ID. Fig. 2 shows a graph where vertices 6 and 8 have different parents that are representatives, namely 3 and 7. Here, the arrows indicate the parent pointers and the dashed line denotes the undirected edge from the original graph that is being “hooked”. After hooking, vertices 6 and 8 have the same representative because vertex 7’s parent pointer was updated to point to vertex 3. However, vertices 6 and 8 do not yet have the same parents, which is where the pointer jumping comes in.

Pointer jumping shortens the path by making the current vertex’s parent pointer point directly to the representative. For example, panel (a) in Fig. 3 shows the paths before and panel (b) after applying pointer jumping to vertex 8. Note that this one pointer-jumping operation not only directly connects vertex 8 to its representative but also compresses the path from vertex 9 to the representative.

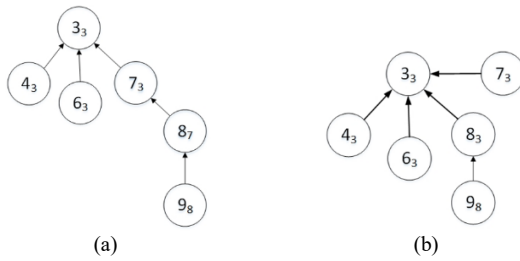


Fig. 3. Example of pointer jumping operation on vertex 8: (a) before, (b) after

Initially, Shiloach and Vishkin’s algorithm considers each vertex as a separate component labelled by its own ID. Then, it repeatedly performs parallel hooking followed by parallel pointer jumping. These two steps are iterated until all vertices of a component have been connected and all paths have been reduced to a length of one. At this point, each vertex points directly to its representative, which serves as the component ID.

Greiner wrote one of the earliest papers on optimizing parallel CC codes [11], including Shiloach and Vishkin’s. Most of the described optimizations are directly or indirectly present in more recent implementations, including in some of the following.

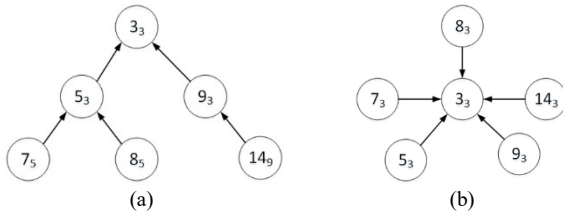


Fig. 4. Example of multiple pointer jumping: (a) before, (b) after

Soman et al. present one of the first GPU implementations of CC [36]. Their algorithm improves upon Shiloach and Vishkin’s as follows. First, instead of operating on the parents of the endpoints of an edge during hooking, they operate on the representatives of the endpoints. Second, they only iterate over the hooking until all vertices in the same component are connected. To make this step faster, they mark edges that no longer need to be processed and can be skipped in later iterations. Third, they introduce “multiple pointer jumping”, which is executed only once after the

hooking is done. In multiple pointer jumping, all parent pointers along the path leading to the representative are updated to directly point to the representative. In other words, the lengths of the paths are reduced to one, as illustrated in Fig. 4. Note that this requires two traversals, one to find the representative and the other to update the parent pointers.

Ben-Nun et al. present Groute, which includes the probably fastest GPU implementation of CC in the current literature [1]. Their algorithm is based on and improves upon Soman’s work as follows. Groute splits the edge list of the graph into $2m/n$ segments and separately performs hooking followed by multiple pointer jumping on each segment. As a result, the hooking and multiple pointer jumping are somewhat interleaved. Moreover, they use atomic hooking, i.e., they lock the representatives of the two endpoints of the edge being hooked, which eliminates the need for repeated iteration.

Wang et al. present the Gunrock library, which also includes a GPU implementation of CC [38]. Their algorithm is again a variant of Soman’s approach. However, instead of processing all vertices and edges in each iteration, Gunrock employs a filter operator. After hooking, the filter removes all edges from further consideration where both end vertices have the same representative. Similarly, after multiple pointer jumping, it removes all vertices that are representatives. In this way, Gunrock reduces the workload after each iteration.

The final GPU implementation of CC we compare to in the result section is IrGL’s, which is interesting because it is automatically generated and optimized from a high-level specification [26]. The algorithm it employs is Soman’s approach.

Ligra is a graph processing framework for shared-memory CPU systems [29]. It provides two implementations for CC, which we call “Comp” and “BFSCC”, the latter of which is one of the fastest multicore CPU implementations of CC in the current literature. The Comp version implements the label-propagation algorithm but maintains a copy of the previous label value in each vertex. This allows it to reduce the workload by only processing vertices whose label has changed in the prior iteration. In contrast, the BFSCC version is based on Ligra’s parallel breadth-first search implementation. It iterates over the vertices, performs parallel BFS on each unprocessed vertex, and marks all reached vertices as belonging to the same component. Ligra+ is an optimized variant of Ligra that internally uses a compressed graph representation, making it possible to fit larger graphs into the available memory [31]. It is otherwise identical to Ligra and contains the same CC implementations. However, Ligra+ is generally faster than Ligra when using its fast compression scheme, which is why we compare to Ligra+.

CRONO is a benchmark suite of graph algorithms for shared-memory multicore machines [1]. Its CC algorithm implements Shiloach and Vishkin’s approach. CRONO’s code is based on 2D matrices of size $n \times d_{max}$, where d_{max} is the graph’s maximum degree. As a consequence, it tends to run out of memory for graphs with high-degree vertices, including some of our graphs.

The Multistep approach was primarily designed to quickly compute *strongly* connected components in parallel, but it also includes a modified implementation for computing CCs [33]. It

starts out by running a single parallel BFS rooted in the vertex with the largest degree, then performs parallel label propagation on the remaining subgraph, and finishes the work serially if only a few vertices are left. The BFS is level synchronous. To minimize overheads, each thread uses a local worklist, which are merged at the end of each iteration.

The work-efficient ndHybrid code is probably the most sophisticated parallel CPU algorithm for CC that we compare to in the result section [30]. It runs multiple concurrent BFSs to generate low-diameter partitions of the graph. Then it contracts each partition into a single vertex, relabels the vertices and edges between partitions, and recursively performs the same operations on the resulting graph.

The final parallel CPU implementation we compare to is from the Galois system, which automatically executes “galoized” serial C++ code in parallel on shared-memory machines [19]. Galois includes three approaches to compute connected components. We only show results for the asynchronous version as we found the other versions to be slower on all tested inputs. This algorithm visits each edge of the graph exactly once and adds it to a concurrent union-find data structure. To reduce the workload, only one of the two opposing directed edges (that represent a single undirected edge) is processed. To run asynchronously and perform union and find operations concurrently, the code uses a restricted form of pointer jumping.

3 THE ECL-CC IMPLEMENTATION

The connected components implementation we describe in this paper is called ECL-CC. It combines the strengths of some of the best algorithms from the literature and augments them with several GPU-specific optimizations. Like Shiloach’s and later approaches, it is based on the union-find technique, keeps a parent pointer in each vertex, and uses a form of hooking and pointer jumping. Like Soman’s and later work, it operates on the representatives of each vertex’s endpoints, skips vertices that do not need processing, and employs an improved version of pointer jumping. Like Groute, it does not require iteration and interleaves hooking and pointer jumping. Like Galois, it is completely asynchronous and lock-free, visits each edge exactly once, only processes edges in one direction, and fully interleaves hooking and pointer jumping. Finally, it incorporates the intermediate pointer-jumping approach by Patwary et al. [27].

The ECL-CC code comprises three phases: initialization, computation, and finalization. Unlike some of the other codes, it is not based on BFS (or DFS) and therefore does not need to find root vertices and does not operate level-by-level. It is recursion free and all phases are fully parallel. It does not have to mark or remove edges. The CPU implementation requires no auxiliary data structure (the GPU implementation uses a double-sided worklist).

The preexisting union-find-based connected-components algorithms initially consider each vertex a separate component labelled by its own ID. This makes the initialization very fast as it requires no computation or memory accesses to determine the initial parent values. However, these values are not particularly good starting points and cause more work in the later computation phase, which can hurt overall performance. As a remedy,

ECL-CC’s initialization phase uses the ID of the first neighbor in the adjacency list that has a smaller ID than the vertex. The vertex’s own ID is only employed if no neighbor with a smaller ID exists. It uses the first neighbor with a smaller ID rather than the neighbor with the lowest ID because the latter requires too many memory accesses to be worthwhile.

ECL-CC makes use of a different type of pointer jumping in the computation phase than any of the GPU implementations described above. Since it is somewhere between (single) pointer jumping and multiple pointer jumping, we call it “intermediate pointer jumping”. Multiple pointer jumping makes all parent pointers encountered during the traversal of a path point directly to the end of the path, i.e., to the representative. In other words, it shortens the path length to one element for all encountered elements. In contrast, single pointer jumping only makes the first parent point to the end of the path. Hence, it compresses the path to one element for the first element but does not shorten the path for any of the later elements. Whereas multiple pointer jumping speeds up future find operations more than single pointer jumping, it requires two passes over the path, one to find the representative and the other to update the elements on the path. Intermediate pointer jumping only requires a single traversal while still compressing the path of all elements encountered along the way, albeit not by as much as multiple pointer jumping. It accomplishes this by making every element skip over the next element, thus halving the path length in each traversal. This is why Patwary et al. call it “path-halving technique” [27]. The code to find the representative of vertex v while performing intermediate pointer jumping is shown in Fig. 5. Since this code does not necessarily reduce all path lengths to one, ECL-CC includes a finalization phase that makes every vertex’s parent point directly to the representative.

```

1: vertex find_repres(vertex v, vertex parent[]) {
2:   vertex par = parent[v];
3:   if (par != v) {
4:     vertex next, prev = v;
5:     while (par > (next = parent[par])) {
6:       parent[prev] = next;
7:       prev = par;
8:       par = next;
9:     }
10:  }
11:  return par;
12: }

```

Fig. 5. Intermediate pointer jumping code

Note that the `find_repres` code is re-entrant and synchronization free even though concurrent execution may cause data races on the parent array. However, these races are guaranteed to be benign for the following reasons. First, the only write to shared data is in line 6. This write updates a single aligned machine word and is therefore atomic. Moreover, it overwrites a valid entry with another valid entry. Hence, it does not matter if other threads see the old or the new value as either value will allow them to eventually reach the representative. Similarly, all the reads of the parent array will either fetch the old or new value, but both values are acceptable. The only problem that can occur is that two threads try to update the same parent pointer at the same time. In

this case, one of the updates is lost. This reduces the code’s performance as duplicate work is performed and the path is not shortened by as much as it could have been, but it does not result in incorrect paths. On average, the savings of not having to perform synchronization far outweighs this small cost. Lastly, it should be noted that the rest of the code either accesses the parent array via calls to the `find_repres` function or changes the parent pointer of a representative vertex but never of a vertex that is in the middle of a path. If the `find_repres` code already sees the new representative, it will return it. Otherwise, it will return the old representative. Either return value is handled correctly.

The GPU version of ECL-CC employs optimizations to reduce load imbalance and to exploit the three levels of hardware parallelism exposed in CUDA. The first level is the threads. The second level is the warps, which are sets of 32 threads that execute in lockstep. The third level is the thread blocks, which hold 256 threads in our implementation. To keep thread divergence and other forms of load imbalance at a minimum, the computation phase of the GPU code is split over three kernels (GPU functions). The first computation kernel operates at thread granularity and only processes vertices up to a degree of 16. Any larger vertices it encounters are placed in a worklist. The second kernel operates at warp granularity and processes vertices with a degree of between 17 and 352, i.e., the processing of the edges of a single vertex is parallelized across a warp. Similarly, the third kernel operates at thread-block granularity and processes vertices with more than 352 edges. These thresholds were determined experimentally. Varying them by quite a bit does not significantly affect the performance. Note that, to save memory space, ECL-CC utilizes a double-sided worklist of size n , which the first kernel populates on one side with the vertices for the second kernel and on the other side with the vertices for the third kernel. This load balancing mechanism is similar to that of Enterprise [23], except we do not need a “small” worklist since we process the low-degree vertices immediately, and our two larger worklists share a single, double-sided worklist.

The core of the edge-processing code, i.e., the hooking operation, is identical in all three computation kernels and shown in Fig. 6. Here, u and v are the two endpoints of the edge and the representative of v is assumed to already be stored in v_rep .

Line 1 ensures that edges are only processed in one direction. Line 6 checks if the representatives of the two endpoints of the edge are the same. If so, nothing needs to be done. If not, the parent of the larger of the two representatives is updated to point to the smaller representative using an atomic CAS operation (either in line 9 or 14 depending on which representative is smaller). The atomicCAS is required because the representative may have been changed by another thread between the call to `find_repres` and the call to the CAS. If it has been changed, u_rep or v_rep is updated with the latest value and the do-while loop repeats the computation until it succeeds, i.e., until there is no data race on the parent. The complete CUDA code is available at <http://cs.txstate.edu/~burtscher/research/ECL-CC/>.

Note that it is sufficient to successfully hook each edge once because, after hooking, both endpoints have the same representative. Any later changes to the parent array by any thread either

shorten some path, which never changes the representative of any vertex, or an existing representative’s parent is made to point to a new representative, which changes the representative of both endpoints to the same new representative. Hence, a previously hooked edge never has to be revisited. The finalization kernel will, ultimately, make all parents point directly to the representative.

```

1: if (v > u) {
2:   vertex u_rep = find_repres(u, parent);
3:   bool repeat;
4:   do {
5:     repeat = false;
6:     if (v_rep != u_rep) {
7:       vertex ret;
8:       if (v_rep < u_rep) {
9:         if ((ret = atomicCAS(&parent[u_rep],
10:                          u_rep, v_rep)) != u_rep) {
11:           u_rep = ret;
12:           repeat = true;
13:         }
14:       } else {
15:         if ((ret = atomicCAS(&parent[v_rep],
16:                          v_rep, u_rep)) != v_rep) {
17:           v_rep = ret;
18:           repeat = true;
19:         }
20:       } while (repeat);
21: }

```

Fig. 6. Core code of the hooking (union) computation

For comparison purposes, we also wrote a parallel CPU implementation of ECL-CC that uses the same general code structure. In particular, it has the same three phases and employs intermediate pointer jumping as well as the improved initialization approach. However, it does not include the GPU-specific optimizations, i.e., it only has a single computation function and requires no worklist. The code is parallelized using OpenMP. In each of the three functions implementing the three phases, the outermost loop going over the vertices is parallelized with a guided schedule. It uses the `find_repres` and hooking code shown in Fig. 5 and Fig. 6, except the atomicCAS is replaced by the gcc intrinsic `__sync_val_compare_and_swap`.

Finally, we also wrote serial CPU code, which is similar to the parallel code but does not contain OpenMP pragmas or atomics. Since there are no calls to atomicCAS that could fail, the do-while loop and its associated variables shown in Fig. 6 are also absent. The serial code still employs intermediate pointer jumping.

4 EXPERIMENTAL METHODOLOGY

We evaluate the connected-components programs listed in Table 1. Where needed, we changed the code that reads in the input graph or wrote graph converters such that all programs could be run with the same inputs. Galois includes three approaches to compute CC. We only show results for the asynchronous version as we found the other versions to be slower on the tested inputs. Similarly, ndHybrid supports OpenMP and Cilk Plus. We only present results for the on average faster Cilk Plus version. Moreover, ndHybrid includes three approaches. We only show results

for the non-deterministic hybrid code, which is faster than the non-hybrid deterministic and non-deterministic versions.

Table 1. The connected-components codes we evaluate

Device	Ser/Par	Name	Version	Source
GPU	parallel	ECL-CC	1.0	[8]
		Groute		[12]
		Gunrock		[13]
		IrGL		[18]
		Soman		[35]
CPU	parallel	CRONO	0.9	[4]
		ECL-CC _{OMP}	1.0	
		Galois	2.3.0	[10]
		Ligra+ BFSCC		[21]
		Ligra+ Comp		[22]
		Multistep		[24]
		ndHybrid		[25]
CPU	serial	Boost	1.62.0	[3]
		ECL-CC _{SER}	1.0	
		Galois	2.3.0	[10]
		igraph		[17]
		Lemon	1.3.1	[20]

In all investigated implementations, we measured the runtime of the CC computation, excluding the time it takes to read in the graphs. In the GPU codes, we also exclude the time it takes to transfer the graph to the GPU or to transfer the result back. In other words, we assume the graph to already be on the GPU from a prior processing step and the result of the CC computation to be needed on the GPU by a later processing step. We repeated each experiment three times and report the median computation time. All ECL-CC implementations verify the solution at the end of the run by comparing it to the solution of the serial code. The verification time is not included in the measured runtime. For all codes, we made sure that the number of CCs is correct.

We present results from two different GPUs. The first GPU is a GeForce GTX Titan X, which is based on the Maxwell architecture. The second GPU is a Tesla K40c, which is based on the older Kepler architecture. The Titan X has 3072 processing elements distributed over 24 multiprocessors that can hold the contexts of 49,152 threads. Each multiprocessor has 48 kB of L1 data cache. The 24 multiprocessors share a 2 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 336 GB/s. We use the default clock frequencies of 1.1 GHz for the processing elements and 3.5 GHz for the GDDR5 memory. The K40 has 2880 processing elements distributed over 15 multiprocessors that can hold the contexts of 30,720 threads. Each multiprocessor is configured to have 48 kB of L1 data cache. The 15 multiprocessors share a 1.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 288 GB/s. We disabled ECC protection of the main memory and use the default clock frequencies of 745 MHz for the processing elements and 3 GHz for the GDDR5 memory. Both GPUs are plugged into 16x PCIe 3.0 slots in the same system. The CUDA driver is 375.26.

For reference, we further show results from two different CPUs. The first system, which hosts the GPUs, has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. Hyperthreading is enabled, i.e., the twenty cores can simultaneously run forty threads. Each core has separate 32 kB L1 caches, a unified 256 kB L2 cache, and the cores on a socket share a 25 MB L3 cache. The host memory size is 128 GB and has a peak bandwidth of 68 GB/s.

The operating system is Fedora 23. The second, older CPU system has dual 6-core Xeon X5690 CPUs running at 3.47 GHz. There is no hyperthreading, i.e., the cores can simultaneously run 12 threads. Each core has separate 32 kB L1 caches, a unified 256 kB L2 cache, and the cores on a socket share a 12 MB L3 cache. The host memory size is 24 GB and has a peak bandwidth of 32 GB/s. The operating system is also Fedora 23.

We compiled all GPU codes with nvcc 8.0 using the “-O3 -arch=sm_52” flags for the Titan X and “-O3 -arch=sm_35” for the K40. The CPU codes were compiled with gcc/g++ 5.3.1 using the “-O3 -march=native” flags.

Table 2. Information about the input graphs

Graph name	Type	Origin	Vertices	Edges*	d _{min}	d _{avg}	d _{max}	CCs
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	2	4.0	4	1
amazon0601	co-purchases	SNAP	403,394	4,886,816	1	12.1	2,752	7
as-skitter	Int. topology	SNAP	1,696,415	22,190,596	1	13.1	35,455	756
citationCiteseer	pub. citations	SMC	268,495	2,313,294	1	8.6	1,318	1
cit-Patents	pat. citations	SMC	3,774,768	33,037,894	1	8.8	793	3,627
coPapersDBLP	pub. citations	SMC	540,486	30,491,458	1	56.4	3,299	1
delaunay_n24	triangulation	SMC	16,777,216	100,663,202	3	6.0	26	1
europa_osm	road map	SMC	50,912,018	108,109,320	1	2.1	13	1
in-2004	web links	SMC	1,382,908	27,182,946	0	19.7	21,869	134
internet	Int. topology	SMC	124,651	387,240	1	3.1	151	1
kron_g500-logn21	Kronecker	SMC	2,097,152	182,081,864	0	86.8	213,904	553,159
r4-2e23.sym	random	Galois	8,388,608	67,108,846	2	8.0	26	1
rmat16.sym	RMAT	Galois	65,536	967,866	0	14.8	569	3,900
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	0	15.7	3,687	428,640
soc-LiveJournal1	j. community	SNAP	4,847,571	85,702,474	0	17.7	20,333	1,876
uk-2002	web links	SMC	18,520,486	523,574,516	0	28.3	194,955	38,359
USA-road-d.NY	road map	Dimacs	264,346	730,100	1	2.8	8	1
USA-road-d.USA	road map	Dimacs	23,947,347	57,708,624	1	2.4	9	1

We used the eighteen graphs listed in Table 2 as inputs. They were obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dimacs) [7], the Galois framework (Galois) [10], the Stanford Network Analysis Platform (SNAP) [34], and the Sparse Matrix Collection (SMC) [37]. Where necessary, we modified the graphs to eliminate loops and multiple edges between the same two vertices. We added any missing back edges to make the graphs undirected. For each graph, Table 2 provides the name, type, origin, number of vertices, number of edges, minimum degree, average degree, maximum degree, and number of CCs. Since the graphs are stored in CSR format, each undirected edge is represented by two directed edges*.

While it may or may not be useful to compute connected components on some of these graphs, we selected this wide variety of inputs to evaluate the tested codes on a broad range of graphs that vary substantially in type and size. The number of vertices differs by up to a factor of 777, the number of edges by up to a factor of 1352, the average degree by up to a factor of 41, and the maximum degree by up to a factor of 53,476.

5 EXPERIMENTAL RESULTS

The following subsections analyze the design of ECL-CC and compare its runtime to that of leading codes from the literature. The main results are normalized to ECL-CC to make them easier to compare as the large disparity in graph sizes yields highly varying runtimes. Values above 1.0 indicate a longer runtime, i.e., all charts show higher-is-worse results. For reference, we also list the absolute runtimes in select cases. All averages refer to the geometric mean of the normalized runtimes.

5.1 ECL-CC Internals

This subsection studies different versions of the three phases of ECL-CC. All results pertain to the Titan X GPU. We report and compare the sum of the runtimes of all kernels, i.e., the total runtime, since changes in one kernel can also affect the amount of work and therefore the runtime of the other kernels.

Fig. 7 shows the normalized runtime of ECL-CC for three different versions of the initialization kernel. The objective of this kernel is to assign a starting value to the label of each vertex. Init1 uses the vertex’s own ID. This is what many implementations from the literature do. Init2 uses the smallest ID of all the vertex’s neighbors. If no neighbor with a smaller ID exists, the vertex’s own ID is used. Init3, which is employed in ECL-CC and represented by the horizontal line at 1.0, uses the ID of the first neighbor in the adjacency list that has a smaller ID than the vertex. If no neighbor with a smaller ID exists, the vertex’s own ID is used.

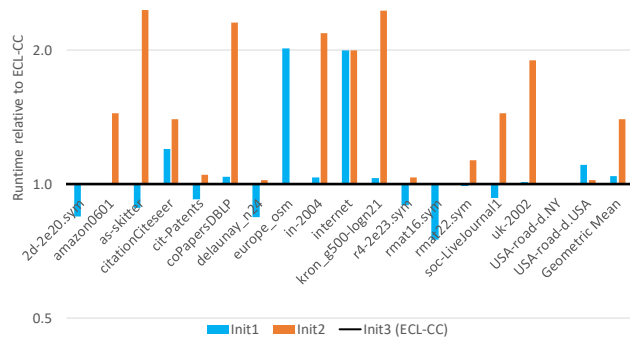


Fig. 7. Relative runtime with different initialization kernels on the Titan X

Init1 can be quite fast in some cases (e.g., 33% faster on rmat16) and requires the fewest operations to determine the initial label value. In other cases, its initialization values are not particularly good starting points and cause more work in later kernels, which is why Init1 can also be rather slow (e.g., over a factor of 2 on europa_osm). On average, it is 4% slower than Init3.

Init2 requires the most operations to determine the initialization value as all neighbors of each vertex must be checked. Consequently, Init2 is never faster than Init3 even though, in some sense, it produces the highest-quality initial values. On average, it is 1.4 times slower.

Init3 is the fastest of the three initialization approaches. It tends to assign better label values than Init1. At the same time, it avoids accessing most neighbors as it stops at the first smaller neighbor. This reduces the number of operations relative to Init2. Whereas Init3 is not much faster than Init1, we chose it for ECL-CC because its worst-case behavior is better (33% slower versus a factor of 2 faster).

Fig. 8 shows the relative runtime of ECL-CC for four different versions of pointer jumping. Jump1 employs multiple pointer jumping, i.e., it makes all elements found along the path point directly to the end of the path. Jump2 uses single pointer jumping, i.e., it only makes the current parent pointer point to the end of

the path. Jump3 does not shorten the path at all but only returns a pointer to the end of the path (i.e., the representative). Jump4, which is used in ECL-CC and represented by the horizontal line at 1.0, implements intermediate pointer jumping, i.e., each element in the path is made to skip the next element, thus halving the path length in each traversal.

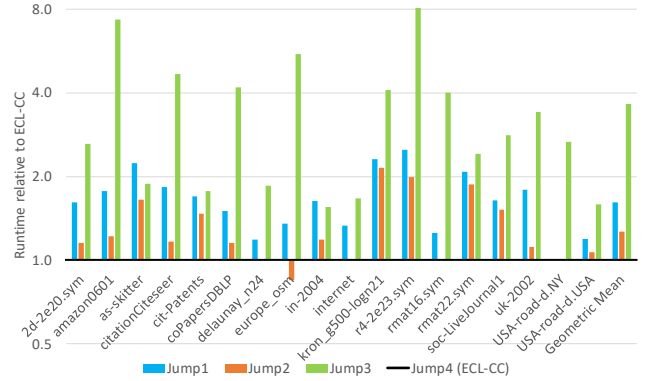


Fig. 8. Relative runtime with different pointer-jumping versions on the Titan X (the cut-off bar extends to 254)

Jump1 (multiple pointer jumping), while yielding the shortest path lengths, is expensive because the paths must be traversed twice, once to find the last element and then a second time to make all the elements point to the last element. On our inputs, it is never faster than Jump4 and, on average, 1.6 times slower.

Jump2 (single pointer jumping) does not require a second traversal but only shortens the path by one element per traversal. In all cases, this results in a lower runtime than Jump1. On europa_osm, it even outperforms Jump4 by 18%. However, on average, Jump2 is 1.26 times slower than Jump4.

As expected, Jump3 (no pointer jumping) performs the worst since it does not shorten the paths. This demonstrates the importance of path compression, even on GPUs where such irregular code can result in significant thread divergence. On average, Jump3 is 3.7 times slower than Jump4.

Jump4 (intermediate pointer jumping), which is used in ECL-CC, is the fastest of the four evaluated approaches on average and in all but one case (europa_osm). It works well because it only traverses the path once while still improving every element along the path and substantially shortening the path length. Moreover, updating nearby neighbors during the traversal has the following two additional benefits. First, and as described in the next paragraphs, it yields good locality because it only updates memory locations that have just been read. Second, it results in the shortest time window between reading and updating the elements on the path, which reduces the chance of the benign data races discussed in Section 3. Since Jump4 has the smallest time window between reading and updating the path elements, the probability for such data races happening is smaller.

To evaluate the locality of the different pointer-jumping approaches, we profiled the whole-application L2-cache accesses on the Titan X. Table 3 lists the relative number of read and write accesses. The results are normalized to ECL-CC (Jump4). Values

above 1.0 indicate more accesses whereas values below 1.0 indicate fewer accesses than ECL-CC. On average, Jump1 (multiple pointer jumping) performs 3.02, Jump2 (single pointer jumping) performs 2.78, Jump3 (no pointer jumping) performs 42.5, and Jump4 (intermediate pointer jumping) performs 8.82 times as many L2 reads as L2 writes.

Table 3. L2 cache read and write accesses relative to Jump4

Graph name	L2 read accesses			L2 write accesses		
	Jump1	Jump2	Jump3	Jump1	Jump2	Jump3
2d-2e20.sym	1.43	1.18	2.18	2.36	1.44	0.25
amazon0601	1.39	1.01	4.73	7.07	6.08	0.48
as-skitter	1.42	0.99	1.40	10.01	9.48	0.38
citationCiteseer	1.45	1.04	2.70	4.96	4.16	0.52
cit-Patents	1.42	1.02	2.22	4.11	3.57	0.56
coPapersDBLP	1.47	1.04	2.52	6.98	6.61	0.40
delaunay_n24	1.23	1.05	1.49	1.82	1.37	0.36
europa_osm	1.96	1.76	9.29	0.83	0.50	0.31
in-2004	1.28	1.00	1.66	2.69	2.37	0.49
internet	1.31	1.14	1.65	1.59	1.20	0.56
kron_g500-logn21	1.71	1.00	2.50	43.38	42.78	0.57
r4-2e23.sym	1.38	1.01	1.81	12.34	9.95	3.35
rmat16.sym	1.50	1.12	3.26	4.00	3.90	1.00
rmat22.sym	1.54	1.00	2.88	9.29	8.53	0.50
soc-LiveJournal1	1.53	1.00	2.68	11.53	10.74	0.44
uk-2002	1.32	1.01	2.33	2.95	2.60	0.45
USA-road-d.NY	1.35	1.30	2.76	0.93	0.75	0.42
USA-road-d.USA	1.31	1.12	1.49	1.64	1.04	0.35
Geometric Mean	1.44	1.09	2.43	4.19	3.45	0.50

These results show that the execution times of the various pointer-jumping approaches correlate with the number of L2 accesses, in particular the more frequent reads. Except for Jump2 on as-skitter, Jump4 always results in the fewest L2 read accesses. Hence, Jump4 must perform fewer reads or have a better L1 hit ratio (i.e., better locality) or both. Furthermore, both multiple pointer jumping and single pointer jumping result in many more L2 write accesses than intermediate pointer jumping. This is interesting because intermediate pointer jumping generally executes more path-compression writes than the other two approaches since it updates pointers in the middle of a path multiple times. Therefore, its L1 write hit ratio must be much higher. Overall, these results indicate that multiple pointer jumping is worse than single pointer jumping on GPUs, which in turn is worse than intermediate pointer jumping, at least for ECL-CC.

For reference, Table 4 lists the average and maximum path lengths during the CC computation. These results need to be taken with a grain of salt, though, as the instrumentation added to obtain them affected the runtimes a little, which likely altered the relative timing among the threads and therefore the path lengths. After all, the path lengths depend on the order in which the vertices and edges are processed and on the possible manifestation of the benign data races.

As the table shows, the paths tend to be very short on average. This is, of course, expected in the presence of any kind of pointer jumping, i.e., path compression. Even the longest observed paths tend to be quite short. The most notable exception is europe_osm. Its average and maximum path lengths are substantially longer than those of the other graphs. Evidently, the processing of this graph happens in a peculiar order that yields abnormally long

paths. Indeed, additional experiments (results not shown) revealed that this input is particularly sensitive to the order in which the vertices are processed.

Table 4. Information about the observed path lengths

Graph name	Average path length	Maximum path length
2d-2e20.sym	1.35	9
amazon0601	1.28	8
as-skitter	1.02	17
citationCiteseer	1.13	11
cit-Patents	1.08	9
coPapersDBLP	1.02	8
delaunay_n24	1.35	13
europa_osm	4.26	122
in-2004	1.14	31
internet	1.49	10
kron_g500-logn21	1.01	6
r4-2e23.sym	1.34	29
rmat16.sym	1.25	10
rmat22.sym	1.05	8
soc-LiveJournal1	1.04	7
uk-2002	1.15	91
USA-road-d.NY	2.62	43
USA-road-d.USA	1.63	27

Fig. 9 shows the relative runtime of ECL-CC for three different versions of the finalization kernel. Since ECL-CC’s computation kernel does not utilize multiple pointer jumping, some parent pointers may end up not directly pointing to the representative. Fini1 employs intermediate pointer jumping to determine the final label values and to compress the paths. Fini2 utilizes multiple pointer jumping. Fini3, which is used in ECL-CC and represented by the horizontal line at 1.0, performs single pointer jumping.

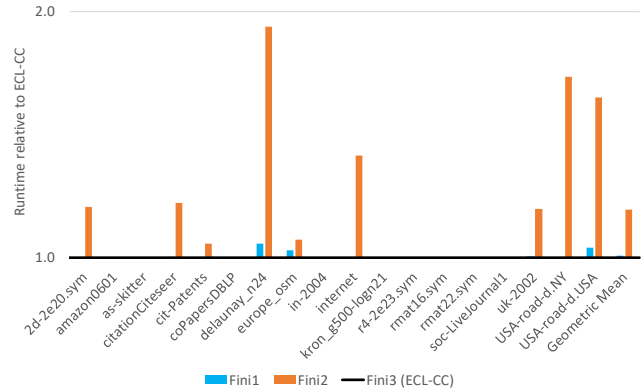


Fig. 9. Relative runtime of different finalizations on the Titan X

On average, there is little difference between Fini1 and Fini3. Whereas Fini1 is never faster, it is also never more than 4% slower on our inputs. On average, it is 0.5% slower. However, Fini2 is significantly slower in several cases as it requires two path traversals. This is especially problematic on delaunay_n24, where it is over 1.9 times slower than Fini3. On average, it is 14% slower. We use Fini3 as it is a little faster and simpler to implement than Fini1.

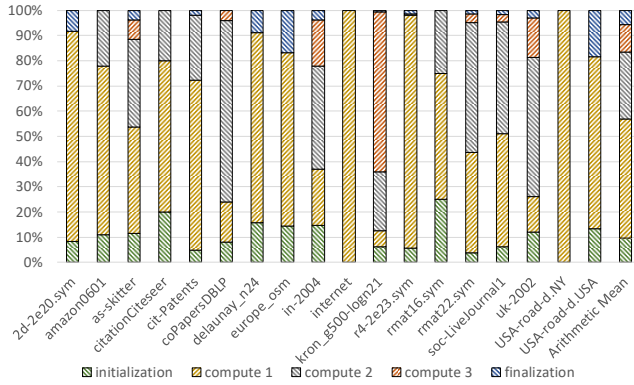


Fig. 10. ECL-CC runtime distribution among the five CUDA kernels on the Titan X

Fig. 10 shows the runtime breakdown of the initialization, the three compute, and the finalization kernels. On average, finalization takes 5.7% of the total runtime, initialization takes 9.8%, compute kernel 3 for the high-degree vertices takes 10.9%, compute kernel 2 for the medium-degree vertices takes 26.5%, and compute kernel 1 for the low-degree vertices takes 47.1%. Overall, 84.5% of the total runtime is spent in the computation phase, making the use of intermediate pointer jumping to speed up the three computation kernels particularly important.

5.2 GPU Runtime Comparison

Fig. 11 shows the relative runtimes on the Titan X GPU with respect to ECL-CC, which is represented by the horizontal line at 1.0. The corresponding absolute runtimes are listed in Table 5. ECL-CC is faster than Gunrock, IrGL, and Soman on all tested inputs. It is also faster than Groute on 15 of the 18 inputs and tied on amazon0601. Groute is 1.25 times faster on as-skitter and 1.34 times faster on europe_osm. On average, ECL-CC is 1.8 times faster than Groute, 4.0 times faster than Soman, 6.4 times faster than IrGL, and 8.4 times faster than Gunrock.

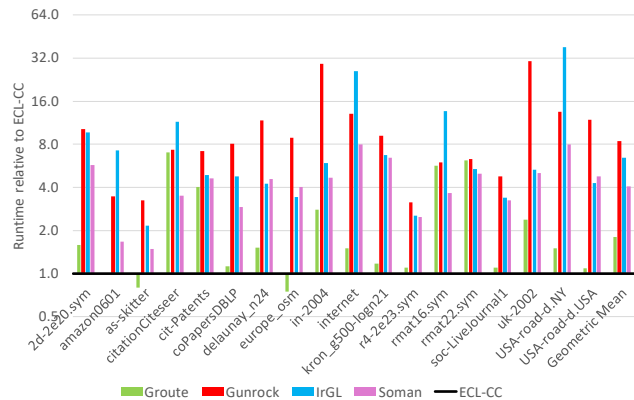


Fig. 11. Titan X runtime relative to ECL-CC

Fig. 12 shows similar results but for the older K40 GPU. As before, ECL-CC is represented by the horizontal line at 1.0. The absolute runtimes are listed in Table 6. Again, ECL-CC is faster than Gunrock, IrGL, and Soman on all tested inputs. It outperforms

Groute on 13 inputs and is tied on USA-NY. Groute is 8% to 32% faster on the remaining four inputs. On average, ECL-CC is 1.6 times faster than Groute, 4.3 times faster than Soman, 5.8 times faster than IrGL, and 11.2 times faster than Gunrock.

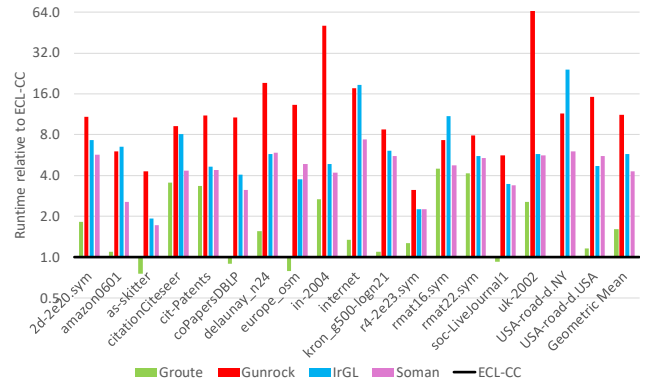


Fig. 12. K40 runtime relative to ECL-CC

Comparing the runtimes listed in Tables 5 and 6, we find that the newer, more parallel, and faster Titan X almost always outperforms the K40. There are only two instances, citationCiteseer and rmat22.sym, where Groute runs faster on the older GPU.

Table 5. Absolute runtimes (in milliseconds) on the Titan X

Graph name	ECL-CC	Groute	Gunrock	IrGL	Soman
2d-2e20.sym	1.2	1.9	12.2	11.6	6.9
amazon0601	0.9	0.9	3.1	6.5	1.5
as-skitter	2.5	2.0	8.1	5.4	3.7
citationCiteseer	0.6	4.2	4.4	6.9	2.1
cit-Patents	14.6	58.6	104.6	70.9	67.6
coPapersDBLP	2.5	2.8	20.1	11.9	7.3
delaunay_n24	14.6	22.1	170.6	62.2	66.7
europe_osm	29.0	21.7	256.9	99.5	116.8
in-2004	2.6	7.3	75.7	15.4	12.1
internet	0.2	0.3	2.6	5.2	1.6
kron_g500-logn21	20.9	24.6	191.8	141.0	134.5
r4-2e23.sym	21.6	23.8	67.6	54.6	53.8
rmat16.sym	0.3	1.7	1.8	4.1	1.1
rmat22.sym	20.5	126.1	129.9	109.5	102.4
soc-LiveJournal1	12.7	14.0	60.4	43.1	41.3
uk-2002	45.8	108.5	1,389.1	243.8	229.7
USA-road-d.NY	0.2	0.3	2.7	7.6	1.6
USA-road-d.USA	14.1	15.3	168.0	60.4	67.1

5.3 Parallel CPU Runtimes

Fig. 13 shows the relative parallel runtimes on the dual 10-core Xeon CPU system. The absolute runtimes are listed in Table 7. ECL-CCOMP, which is represented by the horizontal line at 1.0, as well as Ligra+ BFSCC and ndHybrid are each fastest on five inputs, Ligra+ Comp on two inputs, and Galois on one input. Interestingly, ECL-CCOMP is fastest on the largest inputs, ndHybrid on the medium inputs, and Ligra+ BFSCC on some of the smallest inputs. Except for Multistep, all evaluated codes are faster than ECL-CCOMP on at least one input by between 1.1 and 21 times. However, ECL-CCOMP outperforms each of the other codes by between 8 and 369 times on at least one graph. On average, it is 2% slower than ndHybrid, 1.5 times faster than Ligra+ BFSCC, 2.2

times faster than Ligra+ Comp, 3.5 times faster than CRONO (on the inputs that CRONO supports), 3.6 times faster than Multistep, and 4.7 times faster than Galois.

Table 6. Absolute runtimes (in milliseconds) on the K40

Graph name	ECL-CC	Groute	Gunrock	IrGL	Soman
2d-2e20.sym	2.2	4.0	23.7	16.0	12.5
amazon0601	1.1	1.2	6.6	7.2	2.8
as-skitter	4.1	3.1	17.6	7.9	7.0
citationCiteseer	0.9	3.2	8.3	7.3	3.9
cit-Patents	19.4	64.7	215.0	90.4	84.8
coPapersDBLP	4.5	4.0	48.4	18.3	14.1
delaunay_n24	21.0	32.4	402.6	121.0	123.0
europa_osm	44.2	34.9	584.6	164.5	213.4
in-2004	4.6	12.3	233.9	22.2	19.2
internet	0.3	0.4	5.3	5.6	2.2
kron_g500-logn21	40.1	43.6	351.1	245.2	221.6
r4-2e23.sym	29.8	37.5	93.1	66.7	67.2
rmat16.sym	0.4	1.8	2.9	4.4	1.9
rmat22.sym	27.0	112.2	214.1	150.5	145.7
soc-LiveJournal1	20.1	18.6	113.4	69.8	67.8
uk-2002	83.5	212.9	5,869.2	480.1	466.8
USA-road-d.NY	0.4	0.4	4.6	9.6	2.4
USA-road-d.USA	23.8	27.5	362.4	111.1	132.8

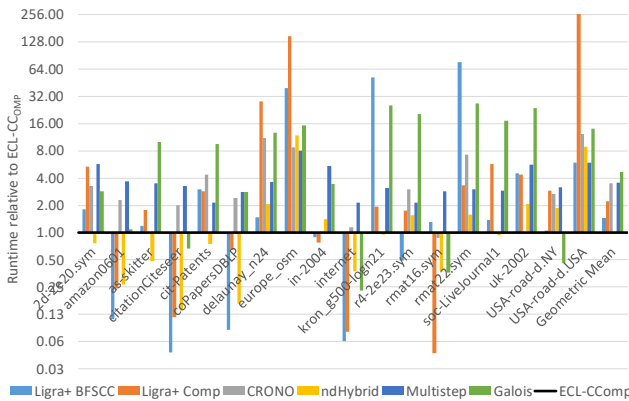


Fig. 13. Parallel E5-2687W runtime relative to ECL-CCOMP

Fig. 14 shows the relative parallel runtimes on the dual 6-core Xeon CPU system. The absolute runtimes are listed in Table 8. ECL-CCOMP, which is represented by the horizontal line at 1.0, is fastest on ten inputs, Ligra+ BFSCC on six inputs, and Ligra+ Comp and Multistep on one input each. Ligra+ BFSCC is faster than ECL-CCOMP by up to 3.4 times, Ligra+ Comp and ndHybrid by up to 1.9 times, and Multistep by up to 1.2 times. However, ECL-CCOMP outperforms each of the other codes by between 16 and 558 times on at least one input. On average, it is 1.7 times faster than Ligra+ BFSCC, 1.9 times faster than ndHybrid, 2.7 times faster than Multistep, 6.8 times faster than CRONO (on the inputs that CRONO supports), 7.2 times faster than Ligra+ Comp, and 22.9 times faster than Galois.

Even though ECL-CCOMP was designed to be a fast CC implementation for GPUs, porting it to OpenMP also yielded a parallel CPU implementation that performs well. On average, it is as fast or faster than the other tested codes. However, there are many inputs on which it is outperformed, especially on the E5-2687W system.

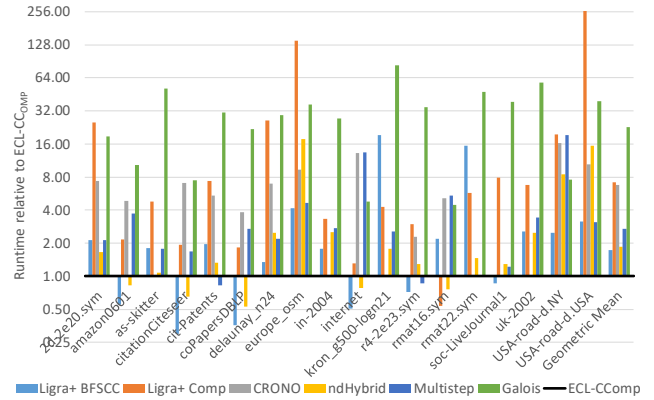


Fig. 14. Parallel X5690 runtime relative to ECL-CCOMP

Table 7. Absolute parallel runtimes (in ms) on the E5-2687W

Graph name	ECL	BFSCC	Comp	CRON	Hybrid	Multi	Galois
2d-2e20.sym	48.9	88.8	262.8	160.7	37.4	278.4	139.5
amazon0601	47.7	5.3	11.6	110.0	12.8	174.9	51.8
as-skitter	63.7	75.6	112.6	n/a	30.8	224.3	641.6
citationCiteseer	55.0	2.6	6.4	109.7	7.6	180.8	36.9
cit-Patents	110.8	333.4	319.1	483.9	83.5	238.3	1,061.0
coPapersDBLP	73.1	6.2	47.1	175.2	11.9	206.8	204.5
delaunay_n24	137.3	202.1	3,883.0	1,532.3	286.0	501.2	1,738.2
europa_osm	178.7	7,046.0	26,540.0	1,574.2	2,130.0	1,438.4	2,753.8
in-2004	52.5	46.5	41.0	n/a	74.3	286.6	181.5
internet	37.7	2.4	3.0	43.1	14.1	81.1	8.7
kron_g500-logn21	117.3	6,117.0	228.0	n/a	111.0	362.6	2,980.2
r4-2e23.sym	119.8	59.0	211.3	363.1	186.0	257.5	2,467.9
rmat16.sym	38.7	51.2	1.8	34.1	14.6	110.4	14.1
rmat22.sym	83.0	6,352.0	274.5	599.2	132.0	250.6	2,212.7
soc-LiveJournal1	89.0	123.1	511.0	n/a	83.9	260.2	1,531.1
uk-2002	165.5	754.5	727.5	n/a	346.0	939.9	3,918.0
USA-road-d.NY	27.8	29.2	81.2	74.1	52.2	87.5	12.8
USA-road-d.USA	117.0	695.7	43,160.0	1,437.4	1,050.0	697.8	1,656.1

Table 8. Absolute parallel runtimes (in millisecond) on the X5690

Graph name	ECL	BFSCC	Comp	CRON	Hybrid	Multi	Galois
2d-2e20.sym	27.0	57.6	680.0	199.3	44.8	57.8	505.3
amazon0601	16.1	8.8	34.9	78.0	13.3	60.2	166.7
as-skitter	29.4	52.9	141.0	n/a	31.6	52.6	1,493.5
citationCiteseer	14.3	4.3	27.7	101.5	9.4	24.1	107.4
cit-Patents	110.1	216.0	813.0	593.4	147.0	91.1	3,427.1
coPapersDBLP	32.8	11.7	59.7	126.2	17.5	88.9	720.3
delaunay_n24	168.8	226.0	4,410.0	1,177.6	422.0	371.2	4,934.5
europa_osm	222.6	926.0	30,800.0	2,087.6	3,930.0	1,039.8	8,127.0
in-2004	24.7	44.2	82.6	n/a	61.9	67.5	677.1
internet	5.1	2.6	6.7	67.6	4.0	68.5	24.3
kron_g500-logn21	114.9	2,230.0	492.0	n/a	206.0	292.5	9,611.0
r4-2e23.sym	182.7	131.0	546.0	420.7	237.0	157.2	6,325.2
rmat16.sym	8.8	19.4	4.7	45.1	6.7	47.9	39.4
rmat22.sym	136.7	2,110.0	779.0	n/a	200.0	140.4	6,507.3
soc-LiveJournal1	123.8	107.0	972.0	n/a	161.0	152.2	4,790.1
uk-2002	207.1	527.0	1,410.0	n/a	516.0	713.0	12,035.9
USA-road-d.NY	5.7	14.1	112.0	93.2	48.1	110.4	43.3
USA-road-d.USA	125.8	398.0	70,200.0	1,316.6	1,950.0	389.3	4,938.2

Comparing the absolute runtimes listed in Tables 7 and 8 yields some unexpected results. Whereas Ligra+ Comp and Galois consistently run faster on the more parallel system, on the remaining codes a third to almost all inputs result in shorter runtimes on the dual 6-core system than on the dual 10-core system. This happens predominantly on the smallest inputs, where the runtime may be too short to overcome the dynamic parallelization over-

head (thread creation and worklist maintenance). Indeed, additional ECL-CC_{OMP} experiments with 20 and 10 instead of 40 threads revealed that 10 threads result in the lowest runtime on the smallest inputs and 20 threads yield the best runtime on most of the medium inputs on the hyperthreaded dual 10-core system. Clearly, some of our inputs are simply too small to scale to 40 OpenMP threads.

5.4 Serial CPU Runtimes

Fig. 15 shows the relative serial runtimes on the E5-2687W CPU. ECL-CC_{SER} is represented by the horizontal line at 1.0. The absolute runtimes are listed in Table 9. ECL-CC_{SER} is the fastest on 16 inputs and Galois on the remaining two, where it is 72% faster on internet and 62% faster on USA-NY. Lemon is also faster than ECL-CC_{SER} on internet but slower than Galois. ECL-CC_{SER} outperforms each of the other codes by between 9.1 and 40 times on at least one input. On average, it is 2.6 times faster than Galois, 5.2 times faster than Boost, 6.7 times faster than igraph, and 9.1 times faster than Lemon.

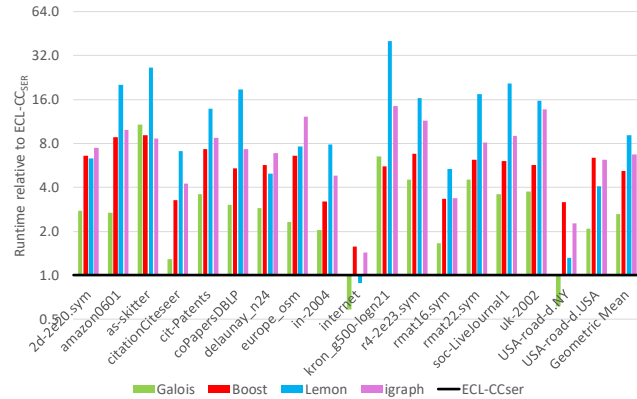


Fig. 15. Serial E5-2687W runtime relative to ECL-CC_{SER}

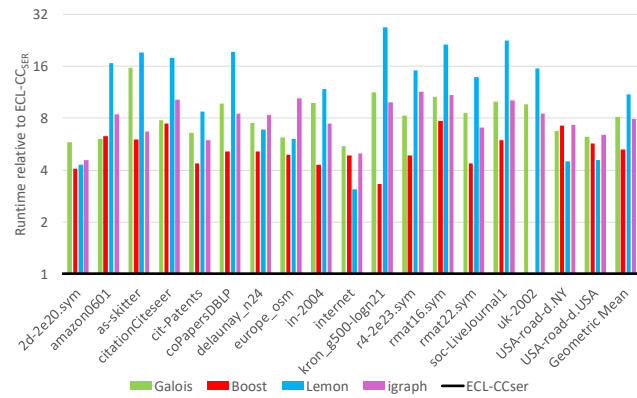


Fig. 16. Serial X5690 runtime relative to ECL-CC_{SER}

Fig. 16 shows the relative serial runtimes on the X5690 CPU. ECL-CC_{SER} is represented by the horizontal line at 1.0. The absolute runtimes are listed in Table 10. ECL-CC_{SER} is fastest on all 18 inputs by at least a factor of 3.1. It outperforms the other codes by between 7.7 and 27 times on at least one input. On average, ECL-

CC_{SER} is 5.3 times faster than Boost, 7.9 times faster than igraph, 8.1 times faster than Galois, and 11 times faster than Lemon.

These results surprised us because ECL-CC was designed for GPUs. Having said that, its good serial performance shows that its unique combination of the best features from various prior CC codes is also useful for creating a fast serial implementation.

Table 9. Absolute serial runtimes (in millisecond) on the E5-2687W

Graph name	ECL-CC _{SER}	Galois	Boost	Lemon	igraph
2d-2e20.sym	42.7	117.9	281.3	268.7	319.2
amazon0601	17.5	46.9	154.6	351.7	174.1
as-skitter	56.2	604.8	511.6	1,487.3	484.9
citationCiteseer	24.4	31.5	79.5	172.2	104.0
cit-Patents	268.8	963.9	1,961.1	3,735.3	2,344.9
coPapersDBLP	66.6	202.5	359.5	1,243.8	488.0
delaunay_n24	510.0	1,464.6	2,913.7	2,545.2	3,516.1
europa_osm	889.8	2,060.1	5,868.5	6,757.3	10,872.6
in-2004	81.8	167.5	261.0	642.6	392.4
internet	9.8	5.7	15.5	8.7	14.0
kron_g500-logn21	447.2	2,911.9	2,486.5	17,998.4	6,458.5
r4-2e23.sym	513.6	2,323.2	3,491.1	8,390.9	5,916.6
rmat16.sym	7.2	12.0	24.1	38.5	24.3
rmat22.sym	458.5	2,072.9	2,835.7	7,976.1	3,721.7
soc-LiveJournal1	405.0	1,453.1	2,458.9	8,361.1	3,655.7
uk-2002	1,004.0	3,773.1	5,732.4	15,731.2	13,728.8
USA-road-d.NY	13.3	8.2	42.2	17.6	30.3
USA-road-d.USA	599.3	1,249.9	3,823.8	2,452.0	3,715.4

Table 10. Absolute serial runtimes (in milliseconds) on the X5690

Graph name	ECL-CC _{SER}	Galois	Boost	Lemon	igraph
2d-2e20.sym	83.7	485.2	340.1	360.8	383.3
amazon0601	26.3	159.7	165.0	438.0	222.2
as-skitter	96.0	1,507.2	576.5	1,845.8	641.9
citationCiteseer	13.2	103.0	98.3	237.2	134.8
cit-Patents	502.1	3,316.0	2,189.8	4,404.1	2,992.1
coPapersDBLP	76.8	747.6	394.7	1,496.8	651.0
delaunay_n24	630.1	4,730.9	3,226.6	4,314.5	5,278.1
europa_osm	1,229.1	7,590.5	6,056.5	7,471.5	12,768.6
in-2004	69.2	677.3	297.5	815.2	514.4
internet	4.1	22.6	19.9	12.7	20.5
kron_g500-logn21	894.1	10,075.9	2,966.6	24,168.4	8,876.0
r4-2e23.sym	720.3	5,945.5	3,499.1	10,906.3	8,195.9
rmat16.sym	3.6	38.2	27.7	77.2	39.1
rmat22.sym	729.9	6,277.4	3,185.2	10,148.0	5,134.9
soc-LiveJournal1	469.1	4,672.9	2,798.9	10,568.1	4,779.0
uk-2002	1,258.6	12,106.1	n/a	19,654.2	10,672.1
USA-road-d.NY	6.1	41.1	44.3	27.5	44.6
USA-road-d.USA	742.5	4,610.9	4,231.1	3,382.5	4,762.3

Comparing the absolute runtimes in Tables 9 and 10, we find that the serial codes generally run faster on the newer system. There are a few exceptions, though, where the older X5690 system is faster. This predominantly happens for the smallest inputs with ECL-CC_{SER}, where the smaller L3 cache size does not matter and the higher clock speed gives the older system an advantage.

5.5 Runtime Comparison across Devices

Fig. 17 compares the geometric-mean Titan X GPU runtimes with the parallel and serial E5-2687W CPU runtimes, all normalized to ECL-CC's runtime on the Titan X, which is represented by the horizontal line at 1.0. These results show that the GPU codes are substantially faster than the CPU codes. Interestingly, the two slowest parallel CPU codes are a little slower than the fastest serial

CPU code. This is known in case of Galois. In case of CRONO, it is an artifact of the averaging because CRONO does not support all inputs. ECL-CC running on the GPU is almost 19 times faster than ndHybrid, the fastest tested parallel CPU code, and 77 times faster than Galois, the fastest tested serial CPU code (not counting ECL-CC_{SER}). Other CPU/GPU pairings would, of course, result in different performance ratios.

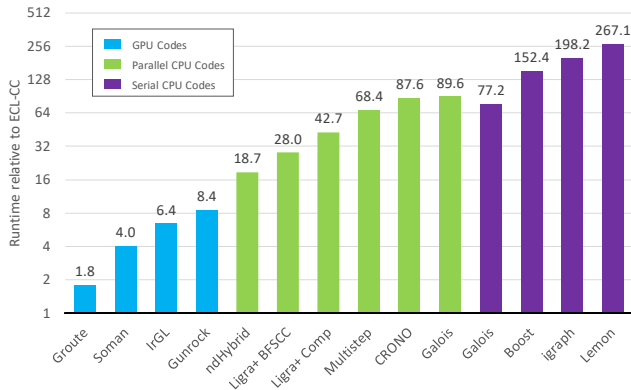


Fig. 17. Geometric-mean runtime across devices relative to ECL-CC running on the Titan X

6 SUMMARY AND CONCLUSIONS

The connected components of an undirected graph are maximal subsets of its vertices such that all vertices in a subset can reach each other by traversing graph edges. Determining the connected components is an important algorithm with applications, e.g., in medicine, computing vision, and biochemistry.

In this paper, we present a new connected-components implementation for GPUs that is faster on average and on most of the eighteen tested graphs than the fastest preexisting CPU and GPU codes. Our approach, called ECL-CC, builds upon the best algorithms from the literature and incorporates several GPU-specific optimizations. It is fully asynchronous and lock-free, processes each undirected edge exactly once, includes a union-find data structure, directly operates on the vertices’ representatives, uses enhanced initialization, employs intermediate pointer jumping, allows benign data races to avoid synchronization, and utilizes a double-sided worklist and three compute kernels to minimize thread divergence and load imbalance. The complete CUDA code is available at <http://cs.txstate.edu/~burtscher/research/ECL-CC/>.

We evaluated ECL-CC on two GPUs and two CPUs from different generations. On the newer devices, our CUDA code is on average 1.8 times faster than Groute, 4.0 times faster than Soman, 6.4 times faster than IrGL, and 8.4 times faster than Gunrock. On average, our OpenMP C++ version performs on par with ndHybrid and is 1.5 times faster than Ligra+ BFSCC, 2.2 times faster than Ligra+ Comp, 3.5 times faster than CRONO, 3.6 times faster than Multistep, and 4.7 times faster than Galois. Our serial C++ code is 2.6 times faster than Galois, 5.2 times faster than Boost, 6.7 times faster than igraph, and 9.1 times faster than Lemon. Whereas there are several cases where other codes outperform ours, those cases typically involve some of the smallest

inputs. In fact, on the largest graph we tested (uk-2002), our GPU and CPU codes yield the lowest runtimes in all instances. On average, our GPU code is 19 times faster than the fastest evaluated parallel CPU code and 77 times faster than the fastest serial code.

Intermediate pointer jumping, a parallelism-friendly technique with good locality to perform path compression in union-find data structures, is probably the most important feature in ECL-CC. It should be able to accelerate other GPU algorithms that are based on union find, such as Kruskal’s algorithm for finding the minimum spanning tree of a graph. In conclusion, we hope our work will accelerate important computations that are based on union-find or connected-components algorithms, thus leading to faster tumor detection, drug discovery, and so on.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback and Sahar Azimi for performing several of the experiments. This work was supported in part by the National Science Foundation under award #1406304 and by equipment donations from Nvidia.

REFERENCES

- [1] Ahmad, M., F. Hijaz, Q. Shi, and O. Khan. “CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores.” *2015 IEEE International Symposium on Workload Characterization*, pp. 44-55, 2015.
- [2] Ben-Nun, T., M. Sutton, S. Pai, and K. Pingali. “Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations.” *22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 235-248, 2017.
- [3] Boost, http://www.boost.org/doc/libs/1_62_0/boost/graph/connected_components.hpp, last accessed on 1/23/2018.
- [4] CRONO, <https://github.com/masabahmad/CRONO>, last accessed on 8/2/2016.
- [5] Csardi G. and T. Nepusz. “The igraph Software Package for Complex Network Research.” *InterJournal, Complex Systems* 1695, 2006.
- [6] Dezsó, B., A. Jüttner, and P. Kovács. “LEMON - An Open Source C++ Graph Template Library.” *Electronic Notes in Theoretical Computer Science*, 264(5):23-45, 2011.
- [7] DIMACS, <http://www.dis.uniroma1.it/challenge9/download.shtml>, last accessed on 1/23/2018.
- [8] ECL-CC, <http://cs.txstate.edu/~burtscher/research/ECL-CC/>, last accessed on 1/23/2018.
- [9] Galler, B.A. and M.J. Fischer. “An Improved Equivalence Algorithm.” *Communications of the ACM*, 7:301–303, 1964.
- [10] Galois, <http://iss.ices.utexas.edu/projects/galois/downloads/Galois-2.3.0.tar.bz2>, last accessed on 1/23/2018.
- [11] Greiner, J. “A Comparison of Parallel Algorithms for Connected Components.” *Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 16-25, 1994.
- [12] Groute, <https://github.com/groute/groute/tree/master/samples/cc>, last accessed on 1/23/2018.
- [13] Gunrock, https://github.com/gunrock/gunrock/blob/master/tests/cc/test_cc.cu, last accessed on 1/23/2018.
- [14] He, L., X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao. “The Connected-Component Labeling Problem: A Review of State-of-the-Art Algorithms.” *Pattern Recognition*, 70:25-43, 2017.
- [15] Hopcroft, J. and R. Tarjan. “Algorithm 447: Efficient Algorithms for Graph Manipulation.” *Communications of the ACM*, 16(6):372–378, 1973.
- [16] Hossam, M.M., A.E. Hassanien, and M. Shoman. “3D Brain Tumor

- Segmentation Scheme using K-Mean Clustering and Connected Component Labeling Algorithms.” *10th International Conference on Intelligent Systems Design and Applications*, pp. 320-324, 2010.
- [17] igraph, <https://github.com/igraph/igraph/blob/master/src/components.c>, last accessed on 1/23/2018.
- [18] IrGL, code obtained from Sreepathi Pai.
- [19] Kulkarni, M., K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. “Optimistic Parallelism Requires Abstractions.” *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 211-222, 2007.
- [20] Lemon, <http://lemon.cs.elte.hu/trac/lemon/wiki/Downloads>, last accessed on 1/23/2018.
- [21] Ligra+ BFSCC, <https://github.com/jshun/ligra/blob/master/apps/BFSCC.C>, last accessed on 1/23/2018.
- [22] Ligra+ Comp, <https://github.com/jshun/ligra/blob/master/apps/Components.C>, last accessed on 1/23/2018.
- [23] Liu, H. and H. H. Huang. “Enterprise: Breadth-First Graph Traversal on GPUs.” *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12, 2015.
- [24] Multistep, <https://github.com/HPCGraphAnalysis/Connectivity>, last accessed on 5/8/2018.
- [25] ndHybrid, <https://people.csail.mit.edu/jshun/connectedComponents.tar>, last accessed on 5/8/2018.
- [26] Pai, S. and K. Pingali. “A Compiler for Throughput Optimization of Graph Algorithms on GPUs.” *2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 1-19, 2016.
- [27] Patwary, M. M. A., P. Refsnes, and F. Manne. “Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure.” *IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 827-835, 2012.
- [28] Shiloach, Y., and U. Vishkin. “An $O(\log n)$ Parallel Connectivity Algorithm.” *Journal of Algorithms*, 3(1):57-67, 1982.
- [29] Shun, J. and G.E. Blelloch. “Ligra: A Lightweight Graph Processing Framework for Shared Memory.” *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 135-146, 2013.
- [30] Shun, J., L. Dhulipala, and G.E. Blelloch. “A Simple and Practical Linear-Work Parallel Algorithm for Connectivity.” *26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 143-153, 2014.
- [31] Shun, J., L. Dhulipala, and G.E. Blelloch. “Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+.” *2015 Data Compression Conference*, pp. 403-412, 2015.
- [32] Siek, J.G., L.-Q. Lee, and A. Lumsdaine. “The Boost Graph Library: User Guide and Reference Manual.” *Addison-Wesley*, 2001. ISBN 978-0-201-72914-6.
- [33] Slota, G.M., S. Rajamanickam, and K. Madduri. “BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems.” *28th IEEE International Parallel and Distributed Processing Symposium*, pp. 550-559, 2014.
- [34] SNAP, <https://snap.stanford.edu/data/>, last accessed on 1/23/2018.
- [35] Soman, <https://github.com/jyosoman/GpuConnectedComponents/blob/master/conn.cu>, last accessed on 1/23/2018.
- [36] Soman, J., K. Kishore, and P. J. Narayanan. “A Fast GPU Algorithm for Graph Connectivity.” *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Ph.D. Forum (IPDPSW)*, pp. 1-8, 2010.
- [37] Sparse Matrix Collection, <https://sparse.tamu.edu/>, last accessed on 1/23/2018.
- [38] Wang, Y., A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J.D. Owens. “Gunrock: A High-performance Graph Processing Library on the GPU.” *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Article 11, 12 pages, 2016.
- [39] Wu, M., X. Li, C.K. Kwok, and S. K. Ng. “A Core-Attachment-Based Method to Detect Protein Complexes in PPI Networks.” *BMC Bioinformatics*, 10(1):169, 2009.