A GPU Algorithm for Detecting Strongly Connected Components

Ghadeer Alabandi
Department of Computer Science
Texas State University
San Marcos, Texas, USA
gaa54@txstate.edu

William Sands
Oden Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, Texas, USA
william.sands@austin.utexas.edu

George Biros
Oden Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, Texas, USA
biros@oden.utexas.edu

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

ABSTRACT

Detecting strongly connected components (SCCs) is an important step in various graph computations. The fastest GPU and CPU implementations from the literature work well on graphs where most of the vertices belong to a single SCC and the vertex degrees follow a power-law distribution. However, these algorithms can be slow on the mesh graphs used in certain radiative transfer simulations, which have a nearly constant vertex degree and can have significant variability in the number and size of SCCs. We introduce ECL-SCC, an SCC detection algorithm that addresses these shortcomings. Our approach is GPU friendly and employs innovative techniques such as maximum ID propagation and edge removal. On an A100 GPU, ECL-SCC performs on par with the fastest prior GPU code on power-law graphs and outperforms it by 7.8× on mesh graphs. Moreover, ECL-SCC running on the GPU outperforms fast parallel CPU code by three orders of magnitude on meshes.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

Strongly connected components, parallelism, performance optimization, GPU implementation

ACM Reference Format:
Ghadeer Alabandi, William Sands, George Biros, and Martin Burtscher. 2023. A GPU Algorithm for Detecting Strongly Connected Components. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607071

1 INTRODUCTION

A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal subset S of its vertices that are all reachable from each other. That is, $\forall a, b \in S$, there exists a directed path both from a to b and from b to a. The subset must be maximal, meaning it cannot be extended by including additional vertices of G. SCCs have the following properties: every vertex $v \in V$ belongs to exactly one SCC, and contracting each SCC into a single vertex turns G into a directed-acyclic graph (DAG). Moreover, every vertex in a non-trivial SCC (i.e., an SCC with more than 1 vertex) has at least 1 adjacent in-neighbor and 1 adjacent out-neighbor in the same SCC.

Finding SCCs is a key building block in many applications, including complex food web analysis [2], data compression [23, 25], finite element simulations [15], and community detection [21]. Due to its significance and the ever-increasing graph sizes, it is important to be able to compute SCCs quickly in parallel.

Tarjan's algorithm [24], a well-known sequential approach for detecting SCCs, runs in linear time in the number of edges and nodes of G. However, it is based on depth-first search, which prevents efficient parallelization. A practical parallel approach called Forward-Backward (FB) was introduced by Fleischer [8]. It randomly selects a pivot vertex and conducts a breadth-first search (BFS) in both the forward and the backward direction starting from the pivot. This partitions the graph into one SCC and three subgraphs that can be processed recursively and in parallel (cf. Section 2 for more details). McLendon [15] added a trim step to boost the performance. The FB-Trim approach starts out by removing (trimming) small SCCs comprising just one or two vertices and then employs the conventional FB algorithm.

Both BFS steps and the recursive subdivision gradually build up parallelism but start with none. This is not a problem on CPUs, where relatively little parallelism is needed to load all cores. However, the initially low parallelism of FB and FB-Trim can be an issue on GPUs that require 100,000s of threads to achieve good performance. The problem is exacerbated by input graphs with high diameters (which lowers the parallelism of BFS) or whose SCCs form a deep DAG (which increases the number of recursive steps).

We present a new approach for detecting SCCs, called ECL-SCC, in which all vertices concurrently act as pivots. Hence, our algorithm has a high degree of parallelism from the start, making it more suitable for devices like GPUs. ECL-SCC assigns two "signature" values to each vertex: (1) the maximum vertex ID on all incoming paths and (2) the maximum vertex ID on all outgoing paths. It then removes all edges from the graph that do not connect vertices with the same signature, which cannot be in the same SCC. The algorithm iterates until no more edges are removed. The final signatures specify to which SCC each vertex belongs.

Although early parallel SCC codes achieve good speedups on synthetic graphs, their performance is limited and sometimes slower than serial code when processing large-scale real-world graphs [4, 14]. This is because real-world graphs, especially those whose vertex degrees follow a power-law distribution (e.g., social networks), have unique features that differ from traditional synthetic graphs: they often contain a single giant SCC and several small SCCs. As a consequence, most recent parallel SCC implementations target graphs with one SCC of size $O(|V|)$ and numerous small SCCs.

In contrast, our work focuses on graphs from thermal radiative transfer simulations, which are widely used to study the behavior of both charged and neutral particle species as well as their interactions with the surrounding media [12]. The radiative transfer equation (RTE) is a hyperbolic partial differential equation that describes the probability of finding a particle belonging to the energy group λ, at the position $\mathbf{x}$, in the direction $\mathbf{\Omega}$, at time t. The RTE can often be efficiently solved by performing "transport sweeps" across multiple (discrete) light propagation directions or *ordinates* [1, 16, 26]. Given an unstructured mesh of the target geometry in which we want to solve the RTE, the sweeping algorithm for each ordinate induces a directed graph whose traversal, starting from nodes with no incoming edges, gives the solution. The occurrence of cycles in these induced graphs can cause significant accuracy, physics, and performance problems [22, 26] that, if not addressed, can lead to livelock during a sweep. Hence, the identification of SCCs is a critical first step in such sweeping algorithms. Note that SCC detection must be performed separately for each discrete ordinate.

In most applications, the resulting meshes tend to have only small SCCs. In the case of linear elements, which are convex, all SCCs are trivial and only encompass a single vertex. However, meshes generated with high-order curved elements tend to produce clusters of small SCCs [10, 15, 22]. Moreover, the SCCs typically form a reasonably deep DAG and the mesh diameters are not small. In other words, the properties of these graphs differ substantially from those targeted by existing parallel SCC codes. Furthermore, none of the existing parallel algorithms for SCC detection in the RTE community target GPUs. We have designed our ECL-SCC approach with these mesh graphs in mind. The method proposed in this work outperforms prior SCC codes by over a factor of 7 on such inputs.

This paper makes the following main contributions.

- We describe a GPU-friendly SCC algorithm called ECL-SCC that is more parallel than prior approaches.
- ECL-SCC is based on a new technique (maximum vertex ID propagation combined with edge removal) and typically detects multiple SCCs concurrently.
- We present domain-specific code optimizations to speed up our CUDA implementation of ECL-SCC.
- On meshes from radiation transport simulations, it outperforms prior approaches by several factors. On other inputs, it performs on par with the fastest codes from the literature.

The latest version of our ECL-SCC CUDA implementation is available in open source through GitHub [5] and on the web [6].

The rest of this paper is organized as follows. Section 2 provides background information and summarizes related work. Section 3 explains our algorithm in detail. Section 4 describes the evaluation methodology. Section 5 presents and discusses the results. Section 6 concludes the paper with a summary.

2 BACKGROUND AND RELATED WORK

In 1972, Robert Tarjan [24] presented the concept of depth-first search and illustrated how it can help improve various graph algorithms. Among these is an algorithm for detecting SCCs, which is perhaps the most well-known serial SCC algorithm. It works by visiting the graph in depth-first search (DFS) order and maintains a stack of the vertices that have not yet been assigned to any component. The algorithm keeps track of the "low-link" value for each vertex. This value is the smallest index of any vertex reachable in one step from that vertex, including itself. The low-link determines which nodes will be removed from the stack to form a new SCC. Upon termination, all nodes will have been visited and all SCCs will have been identified.

Since DFS is difficult to parallelize, several other approaches have been proposed to enable parallel SCC detection. Most of them follow the Forward-Backward (FB) algorithm outlined in the introduction [8]. It performs both a forward and a backward breadth-first search (BFS) starting from a randomly selected pivot to determine two sets of reachable vertices. The intersection of the two sets demarcates the SCC containing the pivot vertex. The remaining vertices are separated into three subgraphs: the vertices that are only in the forward set, the vertices that are only in the backward set, and the vertices that are in neither set. Next, the three subgraphs are independently processed in parallel using the same algorithm, i.e., by selecting a pivot in each of them. Figure 1 shows an example of this procedure.

McLendon [15] improved the parallel FB algorithm by adding a Trim step. This step detects SCCs with one or two vertices. Trimming often reduces the size of the graph significantly, which speeds up the traversals in the BF algorithm. The Trim-1 step identifies SCCs with only one vertex. If a vertex has no in-edges or no out-edges, it is guaranteed to be a trivial SCC of size one. The Trim-2 step identifies SCCs with two vertices. Any two vertices that have no incoming or outgoing edges aside from a bidirectional edge connecting them to each other fall in this category. A Trim-3 step was introduced by Yuede [13] to reduce the size of the graph further. It detects SCCs with 3 vertices based on five patterns. The trim steps can be repeated multiple times because new trivial SCCs might appear after removing other small SCCs from the graph. Figure 2 shows examples of SCCs that can be detected by these trim steps.

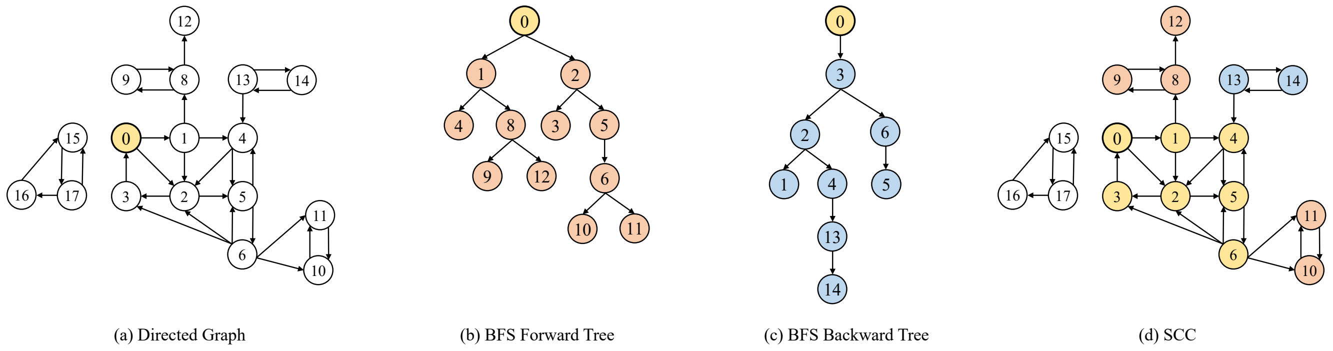

(a) Directed Graph (b) BFS Forward Tree (c) BFS Backward Tree (d) SCC

Figure 1: Example illustrating the steps of the FB algorithm: (a) selecting vertex 0 as the pivot, (b) reachable set of forward BFS, (c) reachable set of backward BFS, (d) resulting SCC (intersection of both sets (yellow)), forward-only set (orange), backward-only set (blue), and unreachable set (uncolored)

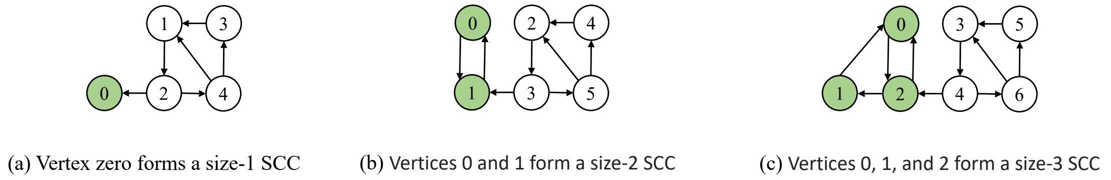

(a) Vertex zero forms a size-1 SCC (b) Vertices 0 and 1 form a size-2 SCC (c) Vertices 0, 1, and 2 form a size-3 SCC

Figure 2: Examples of small SCCs that can be detected by the trim step: (a) a size-1 SCC, (b) a size-2 SCC, and (c) a size-3 SCC

Hong et al. [11] proposed an efficient parallel CPU SCC detection method. This algorithm is one of the first to handle real-world power-law graphs well and uses weakly connected components (WCCs) to detect small SCCs. It is based on the FB algorithm with a trim phase. The authors employ two auxiliary data structures: "mark" and "color". They set the mark flag of a vertex to true once the vertex has been assigned to an SCC. Vertices that belong to the same subgraph have the same color assignment. Furthermore, they mark vertices that have been visited during the forward and backward traversals. Since they target graphs with one giant SCC and many small SCCs, they employ two phases of parallelism, one based on data parallelism to process the single giant SCC and another based on task parallelism to detect the remaining SCCs.

Yuede et al. [13] proposed a different parallel CPU algorithm for identifying SCCs. Their algorithm, called iSpan, uses a spanning tree instead of DFS or BFS to identify the SCCs. To optimize the spanning tree approach, they introduce a new synchronization paradigm called relaxed synchronization (Rsync). Rsync combines both synchronous and asynchronous traversal strategies and is able to not only reduce the amount of synchronization but also to balance the workload. iSpan incorporates two phases. The first phase detects large SCCs using the spanning-tree algorithm. The second phase detects small SCCs of up to size 3 using trim techniques and the remaining small SCCs using the spanning-tree approach. iSpan runs Trim-1 before the large SCC detection and Trim-1, Trim-2, and Trim-3 after the large SCC detection. Yuede et al. implemented iSpan in both OpenMP and MPI.

Barnet et al. [4] present the first GPU algorithm for computing SCCs. Their CUDA implementation is based on the FB algorithm and includes the aforementioned trim and coloring enhancements. Moreover, they designed a new GPU-aware pivot selection approach. It works by having all threads concurrently write the ID of all vertices of a subgraph to a single memory location. The "winning" IDs determine the pivots (one per subgraph).

Li et al. [14] proposed another GPU method for detecting SCCs that is also based on FB with trim. However, the authors parallelized the algorithm by dividing the graph into subgraphs and processing each subgraph simultaneously on different threads. Note that, unlike in the FB algorithm, SCCs can span multiple subgraphs in this algorithm. The algorithm selects several pivots, one for each subgraph. It follows Hong's approach of using different parallelization strategies for detecting large and small SCCs. For large SCCs, Li et al. employ a topology-driven approach with load balancing, whereas for small SCCs, they found load balancing to not be needed.

3 ECL-SCC ALGORITHM AND IMPLEMENTATION

Alg. 1 outlines how ECL-SCC works. It operates on a directed graph with unique vertex IDs and computes two signature values for each vertex v, called v_{in} and v_{out}. At the end of the computation, either value will uniquely identify the SCC to which v belongs. The algorithm comprises an outer loop (Lines 2 to 21) that iterates until the computation has converged. Each iteration goes through three phases: signature initialization, maximum-value propagation, and edge removal. The first phase (Lines 3 to 6) initializes the two signature values of each vertex to the ID of the corresponding vertex. The second phase (Lines 7 to 14) propagates the maximum signature values along the edges. For every directed edge, the *out* value of the source vertex is updated to the *out* value of the destination vertex if it is larger. Similarly, the *in* value of the destination vertex is updated to the *in* value of the source vertex if it is larger. This phase repeats until a fixed point is reached, that is, until no *in* or *out* value changes anymore. The third phase (Lines 15 to 19) removes the edges whose source and destination vertices belong to different SCCs. Detecting this condition is simple. If the signatures of the source and destination differ, the two vertices are guaranteed to not be in the same SCC, and the edge can safely be removed. Then the three phases repeat on the reduced graph that has the same vertices but fewer edges. The algorithm terminates once all vertices have a signature where the *in* value matches the *out* value.

Phase 1 performs $O(|V|)$ work, Phase 2 performs $O(c|E|)$ work, where $c = longest\ cycle\ length$, and Phase 3 performs $O(|E|)$ work. The phases iterate up to d times, where $d = DAG\ depth$ of the DAG that results when contracting each SCC into a single vertex. Thus, the base algorithm performs $O(dc|E| + d|V|))$ work in the worst case. The presence of c explains why our algorithm is more efficient on graphs with small SCCs. We expect the average work complexity to be $O(log(d)log(c)|E| + log(d)|V|)$ for the following reasons. First, if the vertex IDs are randomly distributed, the outer iterations quickly break the DAG into separate pieces (cf. Section 3.2), thus roughly halving the DAG depth in each step and resulting in only $log(d)$ iterations. Our experimental results corroborate this behavior. Second, the cycles can be traversed in as few as $log(c)$ steps using a "path compression" approach (cf. Section 3.3). Since the three phases are parallel (cf. Section 3.4), the expected span of our algorithm, which reflects the length of the longest dependence chain, is $O(log(d)log(c))$.

Algorithm 1 ECL-SCC

```
Input: Directed graph $G = (V, E)$ with unique vertex IDs
1:  $converged \leftarrow false$
2:  while not $converged$ do
                                          ▷ initialize vertex signatures
3:      for all vertices $v \in V$ do
4:          $v_{in} \leftarrow v_{id}$
5:          $v_{out} \leftarrow v_{id}$
6:      end for
                                          ▷ propagate max values
7:      $updated \leftarrow true$
8:      while $updated$ do
9:          for all edges $(u \rightarrow v) \in E$ do
10:             $u_{out} \leftarrow \max(u_{out}, v_{out})$
11:             $v_{in} \leftarrow \max(u_{in}, v_{in})$
12:         end for
13:         $updated \leftarrow$ at least one $v_{in}$ or $u_{out}$ value changed
14:     end while
                                          ▷ remove edges that span SCCs
15:     for all edges $(u \rightarrow v) \in E$ do
16:         if $u_{in} \neq v_{in}$ or $u_{out} \neq v_{out}$ then
17:             $E \leftarrow E \setminus (u \rightarrow v)$
18:         end if
19:     end for
20:     $converged \leftarrow$ all $v_{in} = v_{out}$
21: end while
Output: $\forall v : v_{in}$ (and $v_{out}$) denotes to which SCC vertex $v$ belongs
```

3.1 Algorithm Illustration

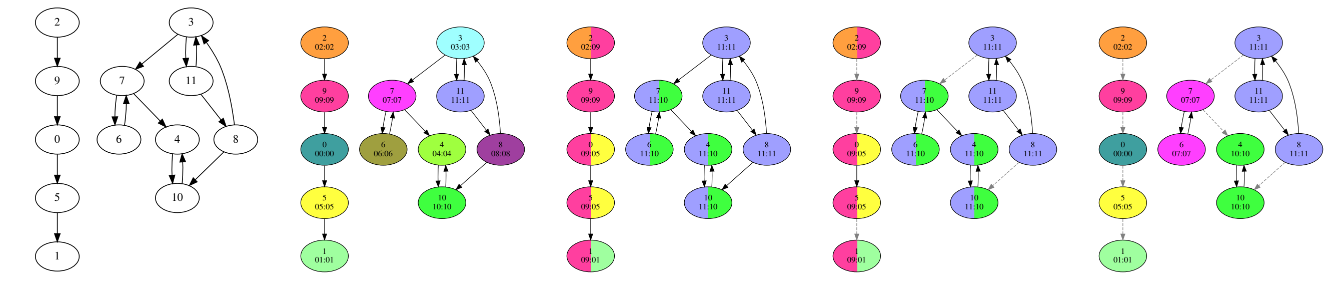
We demonstrate the steps of the ECL-SCC algorithm on the input graph shown in Fig. 3a, which has 12 vertices and 15 edges. Each vertex is labeled with a unique ID between 0 and 11. The graph contains two clusters of vertices that are not reachable from each other. Fig. 3b shows the result of Phase 1, which initializes the two signature values (shown separated by a colon). The vertex color is a function of the signature and not part of the algorithm. We only colored the vertices to make it easier to see which of them have the same signature. Fig. 3c shows the result of Phase 2, that is, after the signature propagation has reached a fixed point. All vertices with a single color are done because their two signature values match. Fig. 3d shows the result of Phase 3, where the removed edges are grayed out and dashed. The ECL-SCC algorithm terminates after repeating these three phases a couple more times. Fig. 3e shows the final signatures and edges. Note how all vertices belonging to the same SCC have the same signature, how each SCC has a different signature, and how all edges within an SCC remain intact whereas all edges between SCCs have been removed.

This example also illustrates how our ECL-SCC algorithm differs from the FB approach described above. Considering only the "left" part of the graph in Fig. 3 that looks like a linked list, picking any of the vertices as the pivot in the FB algorithm will yield either 2 or 3 subgraphs, namely the trivial SCC containing the pivot, all vertices "above" it, and all vertices "below" it. In contrast, ECL-SCC is able to split the linked list into 4 subgraphs in a single step as shown in Fig. 3d, thus potentially speeding up the convergence.

3.2 ECL-SCC Guarantees

3.2.1 Correctness. The SCCs in an input graph are generally clustered together in the sense that some SCCs can be reached from others. A graph may contain multiple such clusters that cannot be reached from each other, as is the case in our example in Fig. 3. One SCC in each cluster must contain the highest-ID vertex of the cluster due to the uniqueness of the vertex IDs. They are vertices 9 and 11 in our example. We refer to the SCCs that contain these vertices as "max" SCCs.

Since, by definition, every vertex in an SCC can be reached by every other vertex in the same SCC and we are computing the maximum reachable ID, all vertices in the max SCCs must end up with the respective highest vertex ID in their v_{in} and v_{out} signature values (after Phase 2). No SCC in a different cluster can have this ID in any of its signatures, no ancestor SCC in the same cluster can have it in any of its v_{in} values, and no descendant SCC in the same cluster can have it in its v_{out} values. Moreover, all ancestor SCCs in the same cluster must have this ID in all of their v_{out} values, and all descendant SCCs in the same cluster must have it in their v_{in} values. Fig. 3c illustrates this. Consequently, all vertices in a max SCC must meet the condition $v_{in} = v_{out}$ and all remaining SCCs must meet the condition $v_{in} \neq v_{out}$. Therefore, each iteration of the ECL-SCC algorithm will concurrently detect at least the max SCC in each cluster and separate them out (meaning all edges to and from those SCCs will be removed in Phase 3 as their signatures do not match). For example, in Fig. 3d, the SCCs "rooted" in vertices 9 and 11 are detected and separated out. The edge removal splits up each cluster into a max SCC and zero or more smaller clusters that contain the



(a) Input Graph **(b) Initialization Step** **(c) Propagation Step** **(d) Edge-Removal Step** **(e) Final Result**

Figure 3: Example illustrating the operation of the ECL-SCC algorithm

remaining vertices. Those smaller clusters will be processed in the next iteration. In this manner, all SCCs will eventually be detected, at which point no clusters will be left and the algorithm stops. Upon termination, all inter-SCC edges will have been removed from the graph and only the intra-SCC edges remain. Moreover, the final signature of each vertex will be the highest ID among all vertices in the same SCC, as shown in Fig. 3e. Due to the uniqueness of the vertex IDs, the signatures uniquely identify the SCCs. Since every vertex can be reached by every other vertex in the same SCC, the signatures of the vertices in an SCC must be identical (after the maximum propagation in Phase 2), even if the SCC is still part of a larger cluster. Therefore, ECL-SCC will never remove an intra-SCC edge, which would incorrectly split an SCC into multiple pieces.

3.2.2 Termination. Since the input graph has a finite number of vertices and edges, Phases 1 and 3 will terminate. Phase 2 also terminates for this reason and because the maximum function is monotonic (i.e., the signature values either stay the same or increase). Consequently, after a finite number of maximum-value propagations, the highest ID among all vertices from which vertex v can be reached will have propagated to v_{in}, and the highest ID among all vertices that are reachable from vertex v will have propagated to v_{out}. This is true for all v. At that point, none of the signatures will change anymore and Phase 2 terminates. The outer loop that iterates over the three phases is also guaranteed to terminate because each iteration will find at least one SCC (cf. Section 3.2.1). Since there cannot be more than $|V|$ SCCs, and $|V|$ is finite, the ECL-SCC algorithm is guaranteed to terminate.

3.3 Performance Enhancements

The baseline algorithm outlined in Alg. 1 can be improved. To boost the performance, we incorporated the following optimizations.

We based our implementation on a worklist that holds the edges of the graph. Hence, Phase 3 does not actually produce a new graph with fewer edges, which might be expensive to accomplish. Instead, it populates a new worklist with just the edges that are still needed. Then, the pointers to the two worklists are swapped to avoid having to copy the new worklist, which would be time intensive.

Another optimization we included is to also remove the edges within the detected max SCCs from the worklist since we do not need them anymore. This reduces the workload as fewer edges will be processed in the following iterations.

Since Phase 2 iterates, it is the most performance critical code and the main target of our optimizations. To accelerate the propagation speed and thus reduce the number of iterations, we employ a technique that is related to "path compression" in union-find data structures [20]. The idea is the following. Assume vertex v has the signature $x : y$ (and recall that the signature values are vertex IDs). Instead of propagating x and y directly to the neighboring vertices, we utilize the corresponding signature values from vertices x and y. Since the signatures are initialized with the vertex ID and can only increase due to the maximum computations, $x_{in} \geq x$ and $y_{out} \geq y$, meaning it is never worse and often better to propagate x_{in} instead of x and y_{out} instead of y. In the code, this amounts to using $in\_max[in\_max[v]]$ instead of $in\_max[v]$. This approach can double the propagation distance in each step, meaning it only takes $O(log(c))$ instead of $O(c)$ time to traverse a cycle of c vertices.

Our implementation takes this idea one step further. Before a signature value s in vertex v is overwritten by a larger value t, our code also checks and conditionally updates the signature of vertex s with value t. This works because all descendants of vertex v (i.e., the vertices reachable from v) share v's ancestors and all ancestors of v (i.e., the vertices from which v is reachable) share v's descendants.

Initially, we implemented each ECL-SCC phase as a separate GPU kernel. However, this resulted in significant launch overhead [19] for inputs where Phase 2 iterates many times. To lower this overhead, we wrote an asynchronous version of the Phase-2 kernel, where every thread block internally iterates until none of the edges assigned to it propagate any more values. This approach can reduce the number of kernel launches by an order of magnitude.

Another optimization that we considered is to use 4 signature values comprising 2 maximums and 2 minimums. The minimums can be computed in the same manner as the maximums except using a minimum operation. This approach would separate out at least 2 SCCs from each SCC cluster in each iteration of Alg. 1. We ended up not using this approach because it doubles the amount of memory needed to store the signatures.

3.4 Parallelization

ECL-SCC does not perform depth- or breadth-first searches, is not recursive, and offers more parallelism because it treats every vertex simultaneously as a pivot. These features makes our approach GPU friendly and easier to parallelize than FB.

Each phase of the ECL-SCC algorithm depends on the result of the prior phase, which is why three global barriers are needed in the body of the outer loop, one after each named code section in Alg. 1. The three "for all" loops are parallel. Phase 1 is embarrassingly parallel, meaning it requires no synchronization. The only synchronization needed in Phase 3 is an atomic add to request a new worklist entry if a thread finds that the edge it is processing is still needed.

Phase 2 can easily be implemented with two atomic max operations. However, as it represents the most performance critical section of our code, we opted for a faster atomic-free implementation [17]. Since the signature propagation is monotonic and any change in a signature value triggers a follow-on iteration, Phase 2 is resilient to temporary priority inversions [18]. This means that all threads with a higher value for a signature can write their value without synchronization. One of the writes will "win", though not necessarily the one with the highest value. However, any written value represents an improvement, and the "losing" thread(s) will try again in the next iteration until the thread with the highest value succeeds. Whereas this may increase the number of iterations needed, it often speeds up the code because no explicit synchronization is performed.

ECL-SCC launches all kernels with 512 threads per block. This number tends to work well on most modern NVIDIA GPUs, which support 1024, 1536, or 2048 threads per streaming multiprocessor (SM). Moreover, we use a persistent-thread approach [9], that is, we launch as many threads as the GPU can concurrently schedule on its SMs (rather than one thread per edge). This means a thread may have to process multiple edges, which is beneficial in combination with our asynchronous Phase-2 implementation.

4 EVALUATION METHODOLOGY

We compared the performance of ECL-SCC with SCC-GPU [14] and iSpan [13], which, respectively, are the fastest GPU and the fastest parallel CPU code we could find. We instrumented each code to measure the SCC computation time, excluding everything else, such as reading in the graph, outputting the result, and verifying the result. Whenever reasonable, we ran each experiment nine times and report the median runtime. Due to very long runtimes, we were only able to run iSpan once on some of the inputs. We verified the solutions of all ECL-SCC runs by comparing them to the results obtained by Tarjan's algorithm. Our primary performance metric is the throughput, which is the number of vertices divided by the runtime. We focus on the throughput because it is a higher-is-better metric, which is more intuitive, and it is normalized by the graph size. This makes the results less input-size dependent.

We evaluated the GPU codes on two generations of NVIDIA GPUs. The first GPU is a Volta-based Titan V with 5120 processing elements distributed over 80 multiprocessors. Each multiprocessor has 96 kB of L1 data cache/shared memory, and the 80 multiprocessors share a 4.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 652 GB/s. The second GPU is an Ampere-based A100 with 6912 processing elements distributed over 108 multiprocessors. Each multiprocessor has 192 kB of L1 data cache/shared memory, and the 108 multiprocessors share a 40 MB L2 cache and 40 GB of global memory with a peak bandwidth of 1555 GB/s.

We also used two systems to evaluate the CPU code. One is AMD-based and the other is Intel-based. The first system has a 16-core 3.5 GHz AMD Ryzen Threadripper 2950X CPU with hyperthreading enabled, allowing the 16 cores to run 32 threads simultaneously. Each core has a 32 kB L1 data cache, a 512 kB unified L2 cache, and all cores share a 32 MB L3 cache. The main-memory size is 48 GB. The second system is based on dual 16-core 2.9 GHz Intel Xeon Gold 6226R CPUs with hyperthreading enabled, allowing the 32 cores to run 64 threads simultaneously. Each core has a 32 kB L1 data cache, a 1 MB unified L2 cache, and the cores on a socket share a 44 MB L3 cache. The main-memory size is 64 GB. Both systems run the Fedora 37 operating system. To compile the GPU codes, we used *nvcc* 12.0 with the "-O3 -arch=sm_70" flags for the Titan V and the "-O3 -arch=sm_80" flags for the A100 GPU. We compiled the CPU code with *gcc/g++* 12.2.1 using the "-O3 -fopenmp" flags.

We used two types of graphs as inputs for our evaluation, the mesh graphs listed in Tables 1 and 2 and the power-law graphs listed in Table 3. The latter graphs were obtained from the SuiteSparse Matrix Collection (SMC) [7]. We selected them because they were also used in prior work [13, 14]. The tables list the name, number of vertices, number of edges, average degree, maximum in-degree, maximum out-degree, number of SCCs, number of size-1 SCCs, number of size-2 SCCs, the size of the largest SCC, and the depth of the DAG formed by the SCCs. For the mesh graphs, some of this information is shown as a minimum and maximum value over two columns since each mesh type includes multiple ordinates. The number of ordinates is denoted by N_Ω, which is equivalent to the number of graphs. We provide additional information about the meshes in Section 4.1.

These tables highlight some important differences between mesh and power-law graphs. For instance, the meshes have only low-degree vertices whereas most of the power-law graphs have some high-degree vertices. Most of the meshes have very small SCCs whereas most of the other graphs have one very large SCC that encompasses the majority of the vertices. Finally, the DAG formed when collapsing the SCCs is quite deep for most of the meshes but shallow for most of the power-law graphs.

To compute the throughputs for the 10 power-law graphs, we used the median runtime of 9 runs. For mesh graphs, we timed the 166 small and 205 large meshes and calculated the throughput based on the average runtime of each mesh group. For example, in Table 1, the beam-hex group comprises 30 mesh graphs. We first measured the average runtime for these 30 graphs and then computed the corresponding throughput based on this average. Note that these runtimes do not include any data transfer to/from the GPU as we are not advocating using GPU code for finding SCCs of graphs stored on the CPU. Rather, we are targeting environments where the graph is already on the GPU from a prior processing step and the SCC result is needed on the GPU for the next processing step.

4.1 Graphs for Radiative Transfer Applications

This subsection describes the graphs that form the basis of any sweeping algorithm for the RTE. Given an unstructured mesh, the sweeping algorithm associates to each ordinate a directed graph that orders the mesh elements to achieve an upwind discretization of the RTE. For an ordinate Ω_d, the computation proceeds

Table 1: Information about the small mesh graphs

Graph	N_Ω	Vertices	Edges	Avg *deg*	Max d_{in}	Max d_{out}	Min SCCs	Max SCCs	Min size-1 SCCs	Max size-1 SCCs	Min size-2 SCCs	Max size-2 SCCs	Min largest SCC size	Max largest SCC size	Min DAG depth	Max DAG depth
beam-hex	30	262,144	769k	2.93	3	3	262,144	262,144	262,144	262,144	0	0	1	1	318	318
star	8	327,680	654k	2.00	2	2	327,680	327,680	327,680	327,680	0	0	1	1	1,534	1,534
torch-hex	32	264,064	782k	2.96	4	5	263,213	263,519	262,551	262,999	504	626	5	8	286	364
torch-tet	32	515,360	1,008k	1.96	3	3	513,410	514,425	511,527	513,501	916	1,847	4	6	484	1,335
toroid-hex	32	196,608	581k	2.96	4	4	189,045	193,745	188,693	193,602	1	15	32	420	220	697
toroid-wedge	32	196,608	486k	2.47	4	4	189,981	193,467	184,625	190,326	3,141	5,524	2	200	282	346

Table 2: Information about the large mesh graphs

Graph	N_Ω	Vertices	Edges	Avg *deg*	Max d_{in}	Max d_{out}	Min SCCs	Max SCCs	Min size-1 SCCs	Max size-1 SCCs	Min size-2 SCCs	Max size-2 SCCs	Min largest SCC size	Max largest SCC size	Min DAG depth	Max DAG depth
klein-bottle	8	8,388,608	19M	2.24	4	4	1	75,750	0	75,746	0	3	8,312,856	8,388,608	1	4
mobius-strip	8	4,194,304	11M	2.98	4	4	758,836	4,194,304	695,463	4,194,304	0	102,243	1	3,246,558	1	15,652
torch-hex	32	2,112,512	6M	2.98	4	5	2,109,019	2,110,311	2,106,301	2,108,211	2,013	2,463	6	16	583	752
torch-tet	32	4,122,880	6M	1.98	3	3	4,113,688	4,117,636	4,104,680	4,112,482	5,092	8,912	4	6	1,019	2,745
toroid-hex	32	1,572,864	5M	2.98	4	4	1,535,516	1,561,334	1,534,396	1,560,997	5	37	64	1,504	444	1,865
toroid-wedge	32	1,572,864	4M	2.48	4	4	1,542,117	1,560,181	1,520,331	1,547,498	12,683	22,539	2	747	570	703
twist-hex	61	6,291,456	19M	3.00	5	5	1	1	0	0	0	0	6,291,456	6,291,456	1	1

Table 3: Information about the power-law graphs

Graph	Vertices	Edges	Avg *deg*	Max d_{in}	Max d_{out}	No. SCCs	Size-1 SCCs	Size-2 SCCs	Largest SCC size	DAG depth
cage14	1,505,785	27,130,349	18.02	41	41	1	1	0	1,505,785	1
circuit5M	5,558,326	59,524,291	10.71	1,290,501	1,290,501	647	15	453	5,555,791	1
com-Youtube	1,134,890	2,987,624	2.63	28,576	4,256	1,134,890	1,134,890	0	1	704
flickr	820,878	9,837,214	11.98	8,549	10,272	277,277	269,944	4,345	527,476	5
Freescale1	3,428,755	18,920,347	5.52	25	27	1,061	1	0	3,408,803	1
Freescale2	2,999,349	23,042,677	7.68	30,478	30,167	55,085	1	54,423	2,888,522	1
soc-LiveJournal1	4,847,571	68,993,773	14.23	13,906	20,293	971,232	947,776	16,875	3,828,682	24
web-Google	916,428	5,105,039	5.57	6,326	456	412,479	399,605	4,169	434,818	34
wiki-Talk	2,394,385	5,021,410	2.10	3,311	100,022	2,281,879	2,281,311	529	111,881	8
wikipedia	3,148,440	39,383,235	12.51	168,685	6,576	1,040,035	1,037,369	2,001	2,104,115	85

by sweeping fluxes through the elements of the mesh with data entering elements through their upwind faces and exiting through downwind faces. The notion of upwind and downwind faces is taken relative to the ordinate Ω_d and is determined by the outward normal vector $\mathbf{n}(\mathbf{x})$ restricted to points along the face of every element. The vertices of the graph represent the elements of an unstructured mesh, while the edges represent a shared face between pairs of neighboring elements. The order in which these elements are traversed forms a directed (possibly cyclic) graph. In the case of high-order curved elements, the normal vector changes its orientation relative to the ordinate at different locations on the same face. This creates the notion of a *re-entrant face*, where neighboring elements are simultaneously upwind of one another, which induces cyclic dependencies in the graph. An example of a high-order mesh with re-entrant faces, which would induce an SCC, is shown in Figure 4.

Our input meshes listed in Tables 1 and 2 were constructed using the MFEM library [3]. We used a collection of meshes, including several sample meshes from MFEM as well as two different representations of the geometry of a plasma torch generated with low-order tetrahedral and hexahedral elements. While the meshes associated with the torch geometry do not contain cycles, the proposed algorithm should also be able to rapidly identify the absence of SCCs. Table 4 provides a summary of the different meshes used in the experiments. Given a mesh and ordinate Ω_d, we construct each graph by iterating through interior faces and extracting the pair of elements (e_1, e_2) that share this face along with the corresponding face transformation. Then, for each point $\mathbf{x}_i$ along this face, we compute the outward unit normal vector $\mathbf{n}(\mathbf{x}_i)$ on e_1. Note that we always use the convention that e_1 points into e_2. Using this normal vector, we can find its orientation relative to the ordinate by checking the sign of the dot product $\Omega_d \cdot \mathbf{n}(\mathbf{x}_i)$. In particular, if $\Omega_d \cdot \mathbf{n}(\mathbf{x}_i) > 0$, then we create an edge pointing from e_1 to e_2. Otherwise, the edge points from e_2 to e_1.

5 RESULTS

In this section, we evaluate the performance of ECL-SCC and compare it to the leading GPU and CPU SCC codes from the literature. We calculated the throughput on each mesh graph based on the average runtime across all ordinates as listed in Tables 5 and 6. For the power-law graphs, we used the runtimes presented in Table 7.

5.1 Throughput Comparison

This subsection compares the performance of ECL-SCC with GPU-SCC, the fastest GPU code from the literature, and iSpan, the fastest parallel CPU code from the literature, on two GPUs and two CPUs. In the result charts, the x-axis lists the inputs and the geometric mean over all of them whereas the y-axis displays the throughput in millions of completed vertices per second. Note that, in some cases, we use a logarithmic scale for the y-axis.

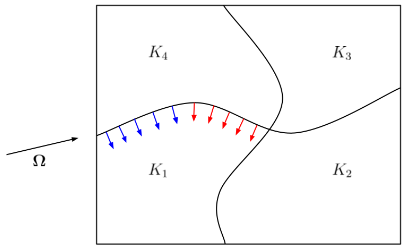


Figure 4: Example of a high-order mesh with re-entrant faces. The normal vectors at the quadrature points along the face shared by elements K_1 and K_4 are shown. The change in color for the normal vectors indicates a change in the sign of the inner product with the ordinate Ω.

Table 4: Properties of the meshes we use in our experiments. They are sample meshes from MFEM [3] along with two different representations of a plasma torch. The twist meshes were built by setting the number of twists to 3 and 6 in the miniapp to change the severity of the distortion.

Mesh Name	Element Type	Order
beam-hex	Hexahedral	1
klein-bottle	Quadrilateral	3
mobius-strip	Quadrilateral	3
star	Quadrilateral	1
torch-hex	Hexahedral	1
torch-tet	Tetrahedral	1
toroid-hex	Hexahedral	3
toroid-wedge	Wedge	3
twist-hex	Hexahedral	3

5.1.1 Small Mesh Graphs. We start by evaluating the performance on the small mesh graphs from Table 1. Figure 5 shows the throughputs on the Titan V GPU. Our ECL-SCC code outperforms GPU-SCC on all mesh groups except for the beam-hex group, where GPU-SCC is 1.12 times faster. This group of meshes contains 30 graphs, all of which have 262,144 SCCs of size 1. Hence, GPU-SCC only uses the Trim-1 phase and not the actual algorithm to process the beam-hex meshes, as it runs the trim phase until all trivial SCCs have been discovered. The same is true for the star meshes. However, the trivial SCCs in these meshes form the deepest DAG of all our small meshes, requiring the Trim-1 phase to be iterated many times. In contrast, ECL-SCC is able to break up the DAG internally, resulting in fewer iterations and faster processing than GPU-SCC. Based on the geometric mean over all small meshes, our code is 6.2 times faster than GPU-SCC on the Titan V.

Figure 6 shows the results of the same experiments but on the A100 GPU. The trends are very similar to those on the Titan V. Again, ECL-SCC outperforms GPU-SCC on all graph groups except beam-hex. However, on the A100, GPU-SCC is only 1.01 times faster on the beam-hex group. Based on the geometric mean over all small meshes, ECL-SCC is 6.5 times faster than GPU-SCC on the A100.

Figure 7 displays the throughputs of ECL-SCC on both GPUs and the throughputs of iSpan on both CPUs. On either GPU, ECL-SCC is

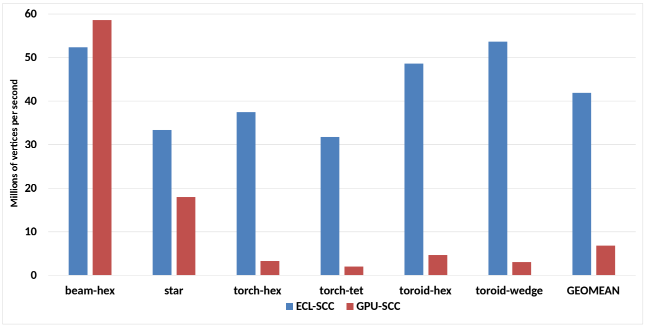


Figure 5: Throughput in millions of completed vertices per second on a Titan V with the small mesh graphs

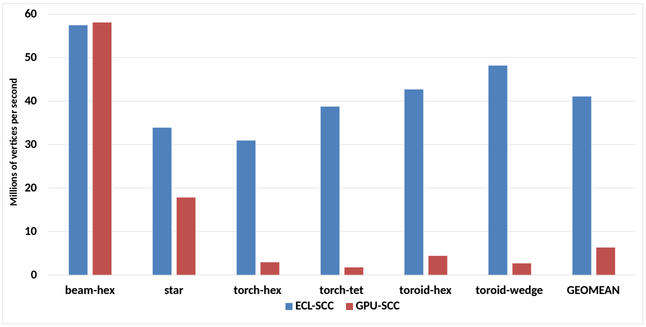


Figure 6: Throughput in millions of completed vertices per second on an A100 with the small mesh graphs

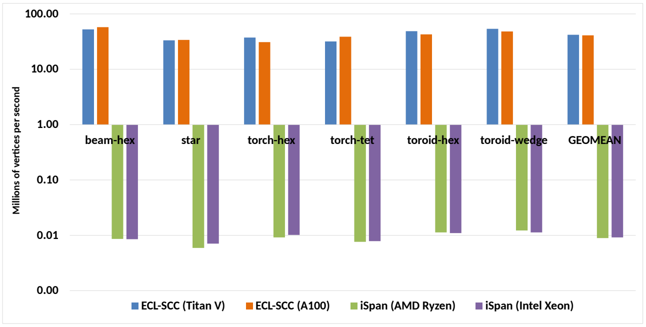


Figure 7: Throughput in millions of completed vertices per second with the small mesh graphs

over three orders of magnitude faster than iSpan running on either CPU for all small mesh graphs. Based on the geometric mean, our code is roughly 4400 times faster than iSpan. Note that iSpan also uses Trim-1 to discover the trivial SCC before detecting any large SCC, but our code still outperforms it even on the mesh graphs with only trivial SCCs such as the beam-hex group.

5.1.2 Large Mesh Graphs. In this subsection, we evaluate the performance on the large mesh graphs from Table 2. Due to the long runtimes, we were only able to run iSpan once per input, and we were unable to run the graphs in the toroid-wedge group to completion (we stopped the runs after 24 hours).

Table 5: Average runtime across ordinates on the small mesh graphs

Graphs	ECL-SCC Titan V	ECL-SCC A100	GPU-SCC Titan V	GPU-SCC A100	iSpan AMD Ryzen	iSpan Intel Xeon
beam-hex	0.0050	0.0046	0.0045	0.0045	30.3428	30.6924
star	0.0098	0.0097	0.0182	0.0184	55.3353	46.1174
torch-hex	0.0071	0.0085	0.0798	0.0901	28.6770	25.9285
torch-tet	0.0162	0.0133	0.2566	0.2939	67.6585	65.3902
toroid-hex	0.0040	0.0046	0.0420	0.0446	17.4289	17.9157
toroid-wedge	0.0037	0.0041	0.0644	0.0729	15.9983	17.4253

Table 6: Average runtime across ordinates on the large mesh graphs

Graphs	ECL-SCC Titan V	ECL-SCC A100	GPU-SCC Titan V	GPU-SCC A100	iSpan AMD Ryzen	iSpan Intel Xeon
klein-bottle	0.2643	0.1564	0.8206	0.6456	0.2524	0.2270
mobius-strip	0.7370	0.3629	3.8899	2.2751	2393.3987	1873.7461
torch-hex	0.0696	0.0376	0.7043	0.4908	2417.4928	1827.9724
torch-tet	0.2772	0.1186	2.4901	1.6397	3977.6657	3746.2860
toroid-hex	0.0345	0.0200	0.3108	0.2524	1569.6603	1210.2917
toroid-wedge	0.0328	0.0189	0.6850	0.5095	1364.8796	> 1 hour
twist-hex	0.1252	0.0586	0.1224	0.1114	0.2203	0.1134

Table 7: Runtime on the power-law graphs

Graphs	ECL-SCC Titan V	ECL-SCC A100	GPU-SCC Titan V	GPU-SCC A100	iSpan AMD Ryzen	iSpan Intel Xeon
cage14	0.0099	0.0046	0.0071	0.0092	0.0362	0.0273
circuit5M	0.0803	0.1077	0.1131	0.1301	0.0946	0.0480
Youtube	0.0035	0.0024	0.0475	0.0574	0.0459	0.0388
flickr	0.0093	0.0079	0.0100	0.0131	0.0124	0.0078
Freescale1	0.0095	0.0043	0.0204	0.0248	0.0843	0.0450
Freescale2	0.0096	0.0061	0.0145	0.0193	crash	crash
soc-LiveJournal	0.0827	0.0254	0.0443	0.0296	0.1123	0.0540
web-Google	0.0084	0.0040	0.0188	0.0216	0.0499	0.0287
wiki-Talk	0.0132	0.0090	0.0126	0.0165	0.0091	0.0059
wikipedia	0.9163	0.2167	0.0861	0.0807	0.0642	0.0334

Figure 8 presents the throughputs on the Titan V. ECL-SCC outperforms GPU-SCC on all graph groups except for twist-hex. This group of meshes contains 61 graphs, all of which consist of a single SCC that contains all vertices. GPU-SCC is faster than our code on 32 of those graphs and 1.02 times faster on average. Note that GPU-SCC was optimized for graphs with one large SCC (and a few small SCCs). This explains why it is a little faster on the meshes of the twist-hex group, which consist of just one SCC. Based on the geometric mean over all large mesh graphs, our code is 6.0 times faster than GPU-SCC on the Titan V.

Figure 9 shows the corresponding throughputs on the A100. Our code outperforms GPU-SCC on all graph groups on this faster and more recent GPU, including by about a factor of 2 on the twist-hex graphs. Based on the geometric mean over all large meshes, ECL-SCC is 8.4 times faster than GPU-SCC on the A100.

Figure 10 shows the throughput of ECL-SCC on both GPUs and the throughput of iSpan on both CPU systems. On the Titan V, it is faster on 5 out of 7 large mesh graphs. On the A100, ECL-SCC is faster than iSpan on all large meshes. On the Titan V, its geometric mean is 1264 and 596 times higher than that of iSpan on the Ryzen and Xeon CPUs, respectively. On the A100, the geometric mean is 2422 and 1142 times higher than iSpan on the Ryzen and Xeon CPUs, respectively. Note that iSpan is also optimized for graphs

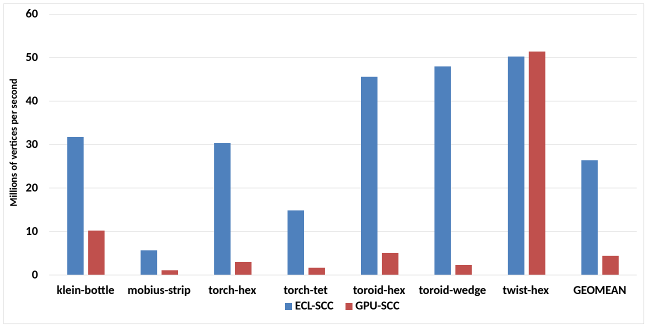


Figure 8: Throughput in millions of completed vertices per second on a Titan V with the large mesh graphs

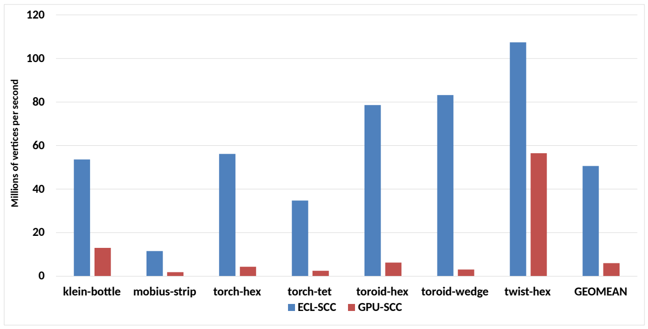


Figure 9: Throughput in millions of completed vertices per second on an A100 with the large mesh graphs

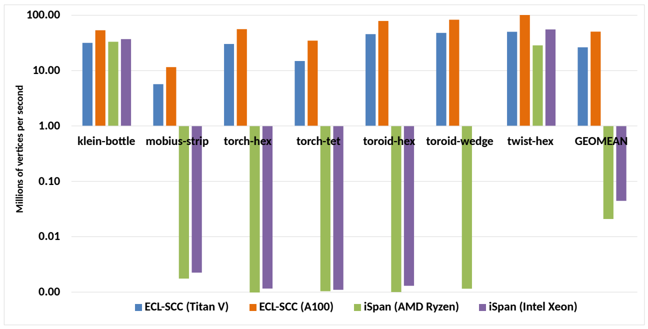


Figure 10: Throughput in millions of completed vertices per second with the large mesh graphs

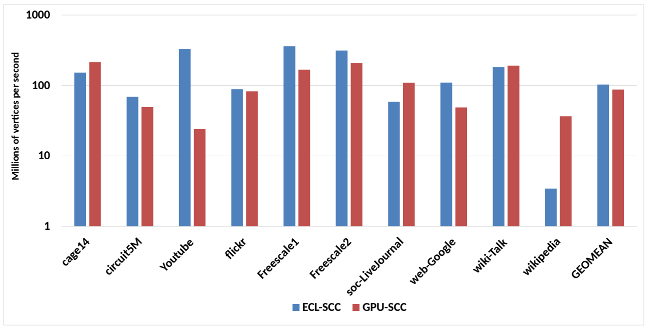


Figure 11: Throughput in millions of completed vertices per second on a Titan V with the power-law graphs

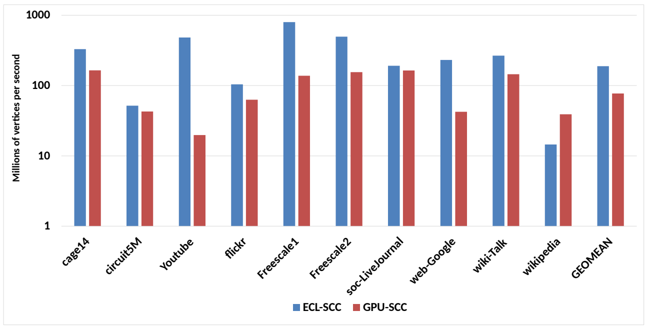


Figure 12: Throughput in millions of completed vertices per second on A100 with the power-law graphs

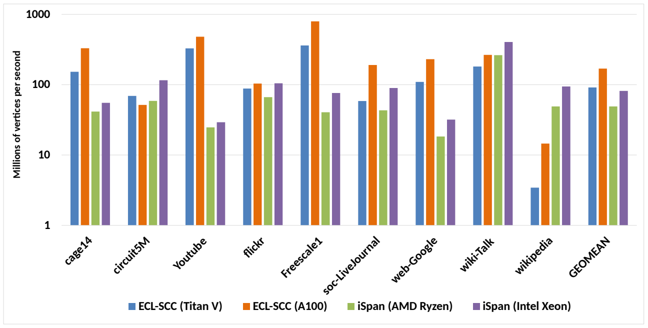


Figure 13: Throughput in millions of completed vertices per second with the power-law graphs

with a power-law distribution, which is why its performance is quite good on the klein-bottle and twist-hex graph groups, which mostly contain graphs with just one large SCC.

5.1.3 Power-law Graphs. Since GPU-SCC and iSpan are optimized for graphs with power-law distributions, we also evaluate how well our code performs on such graphs even though it has been designed for graphs from radiative transfer simulations. Figure 11 displays the resulting throughputs on the Titan V. ECL-SCC outperforms GPU-SCC on 6 of the 10 graphs and has a geometric mean that is 1.18 times higher. Figure 12 shows the throughputs on the A100. ECL-SCC outperforms GPU-SCC on 9 of the 10 graphs and has a geometric mean that is 2.07 times higher than GPU-SCC.

Figure 13 presents the throughputs of ECL-SCC on both GPUs and iSpan on both CPU systems. We encountered a segmentation fault while running iSpan on the Freescale2 graph, which is why this graph is not included in the figure. On the Titan V, ECL-SCC is faster than iSpan on the Ryzen for 7 of the 9 graphs and faster than iSpan on the Xeon for 4 of the 9 graphs. Note that the Xeon has twice as many cores as the Ryzen, which is why iSpan runs faster on the Xeon. ECL-SCC's geometric mean is 1.86 and 1.12 times higher that iSpan's on the Ryzen and Xeon, respectively.

On the A100, ECL-SCC outperforms iSpan on the Ryzen for 6 of the 9 graphs and on the Xeon for 5 of the 9 graphs. ECL-SCC's geometric mean is 3.45 and 2.07 times higher than iSpan's on the Ryzen and the Xeon, respectively. The factors are higher because the A100 is faster than the Titan V on most inputs.

Clearly, the performance advantage of ECL-SCC over GPU-SCC and iSpan is much smaller on this type of graph. In fact, on about half of the inputs, the other codes are faster than ECL-SCC. The reason is that both GPU-SCC and iSpan are optimized for power-law graphs whereas ECL-SCC is optimized for another kind of graph with very different properties. Nevertheless, our code is still competitive even on power-law graphs, highlighting its versatility.

5.1.4 Expanded Meshes. The speedup trends for the small and large meshes are similar, and the mesh sizes we use are typical for RTE applications. However, a significant portion of these meshes fits in the last-level caches of our CPUs and GPUs. To see whether the speedup trends also hold for much larger meshes, we took

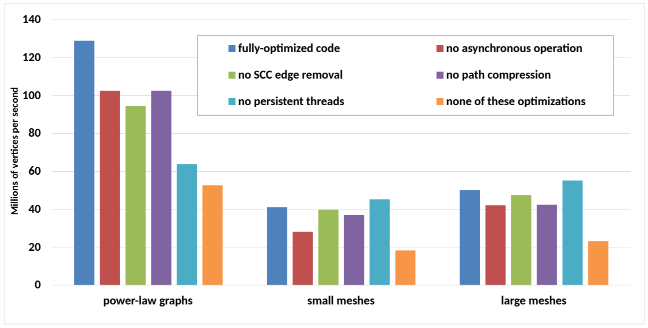


Figure 14: A100 performance impact when removing certain code optimizations from ECL-SCC

the first ordinate of our twist-hex and large toroid-hex meshes and replicated them to create 10-times larger inputs. The expanded twist-hex mesh has 62,914,551 vertices, 188,937,390 edges, and 1 SCC. The expanded toroid-hex mesh has 15,728,631 vertices, 46,841,420 edges, and 15,581,611 SCCs. We selected these two meshes because the former is the mesh where GPU-SCC and iSpan perform the best and the latter is representative of our RTE meshes.

Evaluating these much larger inputs on the A100 GPU and Xeon CPU, we found the speedup trends to still hold. Specifically, on the expanded twist-hex mesh, ECL-SCC took 0.697 seconds, iSpan running with 64 threads on the CPU took 1.4× longer, and GPU-SCC crashed because the mesh is too large. On the expanded toroid-hex mesh, ECL-SCC took 0.173 seconds, GPU-SCC took 78.5× longer, and iSpan timed out after 3 hours. In summary, ECL-SCC performs a little better than the fastest code from the literature on a very large mesh with a single SCC and is much faster on a very large mesh with many small SCCs.

5.2 Optimization Evaluation

This subsection evaluates several code optimizations in ECL-SCC (cf. Section 3.3). The first optimization is implementing Phase 2 in an asynchronous manner to reduce the number of kernel launches. The second optimization is removing all completed edges from the graph and not just the ones that span SCCs to reduce the workload in later iterations. The third optimization is adding path compression so the labels can be propagated faster and not only to the direct vertex neighbors. The fourth optimization is to employ persistent threads, meaning multiple edges and vertices are assigned to a single thread, so that fewer thread blocks need to be launched.

Fig. 14 shows the results for our three types of inputs. For brevity, we only display results for the A100. Each set of bars represents the geometric-mean throughput over all graphs in that category. The individual bars show the throughput of the ECL-SCC code with all optimizations included, the throughput when disabling any one of the studied optimizations, and the throughput when all four optimizations are disabled together.

Path compression and especially asynchronous operation help on all three types of input graphs. Removing the SCC edges helps only marginally on the meshes but quite a bit on the power-law graphs. This is expected as most of the meshes only contain small SCCs and, therefore, only few intra-SCC edges. Conversely, in many of the power-law graphs, most of the edges lie within an SCC and removing them saves significant computation.

Using persistent threads, i.e., assigning multiple edges to a single thread, greatly boosts the performance on the power-law graphs but actually hurts on the mesh graphs. This is the case because persistent threads can speed up the code by improving the effectiveness of the asynchronous execution, but they can also slow down the code because they always process all assigned edges, even when some edges no longer need to be processed. If the edges assigned to a thread belong to the same SCC, as is often the case in power-law graphs, they all need to be processed for the same number of iterations, meaning the benefit outweighs the downside. However, if the edges belong to different SCCs whose maximum-value propagations reach a fixed point after a different number of iterations, as is often the case in meshes, the downside outweighs the benefit. We included this "optimization" regardless to make ECL-SCC competitive on power-law graphs. However, when only targeting mesh graphs with small SCCs, it should be removed, which would yield a 10% performance boost.

Disabling all four optimizations results in the lowest performance. On all three graph types, these optimizations together more than double the throughput of ECL-SCC, highlighting the importance of including them.

6 SUMMARY

The parallel detection of strongly connected components (SCCs) is an important algorithm for graph analysis and has broad applications to fields in scientific computing. Most existing parallel SCC algorithms are based on the Forward-Backward (FB) algorithm and generally perform well on power-law graphs, in which a few vertices have a very high degree and most vertices tend to belong to one large SCC. In contrast, this paper introduces a novel GPU algorithm, called ECL-SCC, to detect SCCs in graphs arising from radiative transfer applications, which typically only contain low-degree vertices and whose SCCs tend to be very small. Unlike the FB algorithm, our algorithm simultaneously uses every vertex as a pivot, which increases parallelism and enables it to partition the graph more quickly. Additionally, it employs new techniques such as maximum vertex-ID propagation and edge removal. Our CUDA implementation incorporates several domain-specific code optimizations, including combining asynchronous operation with persistent threads, utilizing path compression, and employing data-driven edge-based processing. We evaluated our GPU code and optimizations on several types of input graphs. Experimental results on two GPUs from different generations and two CPUs from different vendors demonstrate that the performance of our algorithm is commensurate with the fastest preexisting codes on power-law graphs and outperforms them by over a factor of 5 on average on mesh graphs.

ACKNOWLEDGMENTS

This work has been supported in part by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969, by the National Science Foundation under Awards 1955367 and 2204226, and by equipment donations from NVIDIA Corporation.

REFERENCES

[1] Marvin L. Adams and Edward W. Larsen. 2002. Fast iterative methods for discrete-ordinates particle transport calculations. *Progress in Nuclear Energy* 40 (2002), 3–159. Issue 1.

[2] Stefano Allesina, Antonio Bodini, and Cristina Bondavalli. 2005. Ecological subsystems via graph theory: the role of strongly connected components. *Oikos* 110, 1 (2005), 164–176.

[3] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, et al. 2021. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications* 81 (2021), 42–74.

[4] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Češka. 2011. Computing strongly connected components in parallel on CUDA. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 544–555.

[5] Martin Burtscher. 2023. ECL-SCC Git Repository. https://github.com/burtscher/ECL-SCC. Accessed: 2023-08-18.

[6] Martin Burtscher. 2023. ECL-SCC Website. https://cs.txstate.edu/~burtscher/research/ECL-SCC/. Accessed: 2023-08-18.

[7] Tim Davis. [n. d.]. SuiteSparse Matrix Collection. http://sparse.tamu.edu, Last accessed on 2023-03-16.

[8] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing: 15 IPDPS 2000 Workshops Cancun, Mexico, May 1–5, 2000 Proceedings 14*. Springer, 505–511.

[9] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

[10] T.S. Haut, P.G. Maginot, V.Z. Tomov, B.S. Southworth, T.A. Brunner, and T.S. Bailey. 2019. An efficient sweep-based solver for the S_N equations on high-order meshes. *Nuclear Science and Engineering* 193 (2019), 746–759. Issue 7.

[11] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[12] John R. Howell, M. Pinar Julian Mengüc, Kyle Daun, and Robert Siegel. 2020. *Thermal Radiation Heat Transfer* (7 ed.). Taylor & Francis.

[13] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 731–742. https://doi.org/10.1109/SC.2018.00061

[14] Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang. 2017. High performance detection of strongly connected components in sparse graphs on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. 48–57.

[15] William Mclendon III, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel and Distrib. Comput.* 65, 8 (2005), 901–910.

[16] Warren F. Miller and Elmer E. Lewis. 1993. *Computational methods of neutron transport*. Wiley.

[17] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-Free Irregular Computations on GPUs *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2458523.2458533

[18] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 456–471. https://doi.org/10.1145/2517349.2522739

[19] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015

[20] Md. Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. 2012. Multi-Core Spanning Forest Algorithms Using the Disjoint-Set Data Structure. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, USA, 827–835. https://doi.org/10.1109/IPDPS.2012.79

[21] Wen-Chih Peng, Haixun Wang, James Bailey, Vincent S Tseng, Tu Bao Ho, Zhi-Hua Zhou, and Arbee LP Chen. 2014. *Trends and Applications in Knowledge Discovery and Data Mining: PAKDD 2014 International Workshops: DANTH, BDM, MobiSocial, BigEC, CloudSD, MSMV-MBI, SDA, DMDA-Health, ALSIP, SocNet, DMBIH, BigPMA, Tainan, Taiwan, May 13-16, 2014. Revised Selected Papers*. Vol. 8643. Springer.

[22] Steven J. Plimpton, Bruce Hendrickson, Shawn P. Burns, William McLendon III, and Lawrence Rauchwerger. 2005. Parallel S_n sweeps on unstructured grids: algorithms for prioritization, grid Partitioning, and cycle detection. *Nuclear Science and Engineering* 150, 3 (2005), 267–283.

[23] K.H. Randall, R. Stata, R.G. Wickremesinghe, and J.L. Wiener. 2002. The Link Database: fast access to graphs of the Web. In *Proceedings DCC 2002. Data Compression Conference*. 122–131. https://doi.org/10.1109/DCC.2002.999950

[24] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[25] Sebastiaan J. van Schaik and Oege de Moor. 2011. A Memory Efficient Reachability Data Structure through Bit Vector Compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 913–924. https://doi.org/10.1145/1989323.1989419

[26] Jan I.C. Vermaak, Jean C. Ragusa, Marvin L. Adams, and Jim E. Morel. 2019. Massively parallel transport sweeps on meshes with cyclic dependencies. *J. Comput. Phys.* 425 (2019), 109892.