arXiv:2103.15217v1 [cs.DS] 28 Mar 2021

Euler Meets GPU: Practical Graph Algorithms with Theoretical Guarantees*

Adam Polak, Adrian Siwiec, and Michał Stobierski

Jagiellonian University

Abstract

The Euler tour technique is a classical tool for designing parallel graph algorithms, originally proposed for the PRAM model. We ask whether it can be adapted to run efficiently on GPU. We focus on two established applications of the technique: (1) the problem of finding lowest common ancestors (LCA) of pairs of nodes in trees, and (2) the problem of finding bridges in undirected graphs. In our experiments, we compare theoretically optimal algorithms using the Euler tour technique against simpler heuristics supposed to perform particularly well on typical instances. We show that the Euler tour-based algorithms not only fulfill their theoretical promises and outperform practical heuristics on hard instances, but also perform on par with them on easy instances.

1 Introduction

Over the last two decades graphical processing units (GPU) proved very successful for general purpose computation. Their architecture makes them particularly well suited for highly parallel jobs such as matrix multiplication and other linear algebraic operations. Hence, the recent neural networks boom further increased the GPU popularity. While matrix multiplication is an embarrassingly parallel problem – and solving it efficiently with GPU requires more of an engineering than algorithmic insight – for other less regular problems we face an obstacle: the GPU architecture is complex and has no good theoretical model.[1] In order to design an effective GPU algorithm, performing well in practice, one needs to take care of certain low-level engineering aspects which cannot be abstracted away, e.g., different synchronization at different levels of parallelization (i.e. *threads* and *blocks*), or high latency global memory, which requires overlapping reads to achieve a good throughput.

Probably the closest model is the *parallel random-access machine* (PRAM) model, developed in the 1980s. Certain features make it though extremely unrealistic, e.g. zero-cost synchronization, or zero-latency memory. It thus serves more as a source of inspiration for GPU algorithms rather than an actual model. Some PRAM algorithms were successfully adapted to GPU (e.g. prefix sum [51]), but some are believed to be too complex and to have too large constant overheads to become practical on GPU (see e.g. [19, 61] on the biconnectivity algorithm of [58]).

A common approach for GPU algorithm design is to use simple low-overhead algorithms that exploit parallelism assumed to be present in typical real-world instances (but not guaranteed by a combinatorial structure of the problem). This is exemplified by arguably the most popular GPU graph primitive, *breadth-first search* (BFS). GPU implementations of BFS (e.g. [39, 63])

*This work was partially supported by the Polish Ministry of Science and Higher Education grant DI2012 018942 and National Science Center of Poland grant 2017/27/N/ST6/01334.

[1]Such as the word-RAM model, which proved to be a good model for modern CPU, in the sense that it often facilitates developing practical algorithms.

are extremely efficient on graphs with small diameter – which is the case for many real-world graphs – but become prohibitively slow on, say, paths.

The Euler tour technique is a classical tool for designing parallel graph algorithms, originally proposed for the PRAM model [58]. We ask whether it can be adapted to run efficiently on GPU. We focus on two established applications of the technique: (1) the problem of finding *lowest common ancestors* (LCA) of pairs of nodes in trees, and (2) the problem of finding *bridges* in undirected graphs. In our experiments, we compare theoretically optimal algorithms using the Euler tour technique [50, 58] against simpler heuristics [38, 61], which are state-of-the-art GPU algorithms, supposed to perform particularly well on typical instances. We show that the Euler tour-based algorithms not only fulfill their theoretical promises and outperform practical heuristics on hard instances, but also perform on par with them on easy instances.

1.1 Related Work

Euler Tour

Tarjan and Vishkin [58] first introduced the Euler tour technique, and applied it to design a parallel biconnectivity algorithm. Later the technique was used in many PRAM algorithms from different areas, e.g. algorithms on trees [14,23], planar graphs [22,44], polygons [24], texts [12,16], and context-free grammars [47]. It also found applications beyond PRAM, e.g. in large-scale distributed graph processing [7, 65], and dynamic problems [28, 41, 57].

Graph Algorithms on GPU

First attempts to speed up graph algorithms using GPU included transitive closure [34] and APSP [40]. Harish et al. [26,27] took a systematic approach and developed GPU implementations of BFS, ST-connectivity, shortest paths, minimum spanning tree, and max-flow algorithms. Other examples include triangle counting [62], connected components [31, 54], and strongly connected components [5, 37]. Most of those works report speedups in the range of 10-100x over single-core CPU baselines. They also make it clear that developing from scratch an efficient GPU algorithm, which can outperform a baseline, often requires a significant amount of nontrivial fine-tuning. For that reason, there is a growing interest in graph processing libraries for GPU, delivering highly-optimized building blocks allowing an easy construction of efficient algorithms, e.g. Medusa [66], Gunrock [63], Groute [8].

LCA in Trees

Aho, Hopcroft and Ullman [1] were the first to study the LCA problem. On top of being a natural combinatorial problem, LCA was also extensively studied as a subproblem emerging in other graph problems, e.g., dominator tree in a directed flow-graph [1], maximum weighted matching [21], network routing [59], phylogenetic distance computation [38]. Harel and Tarjan [25] first gave an optimal LCA algorithm, with linear time preprocessing and constant time queries. It was subsequently simplified by Schieber and Vishkin [50], and Bender and Farach-Colton [9]. The approach of Schieber and Vishkin [50] is not only simple but also naturally parallel, yielding an optimal, $O(\log n)$-time and $O(n)$-work, PRAM algorithm.

Martins et al. [38] implemented an LCA algorithm on GPU. They opted for a naïve approach, working well on shallow trees, but with a linear, instead of a constant, worst-case query time. To the best of our knowledge, the only GPU alternative to their algorithm is the work of Soman et al. [55], which focuses on discrete range searching primitives, and mentions LCA only briefly as one of applications. They assume, however, that a required preprocessing is already done and the necessary data structures are given in input, and they focus only on an algorithm for answering queries. Therefore, we cannot compare directly with their approach.

Bridge-Finding and Biconnectivity

The algorithmic study of biconnectivity dates back to Hopcroft and Tarjan [29, 30] and Paton [43], who presented linear-time algorithms, based on depth-first search, for finding articulation points, bridges, and 2-connected components in undirected graphs. These graph-theoretic concepts are central in, e.g., planar graph recognition and drawing [33], or network analysis [10, 48].

Since depth-first search is inherently sequential [45], a parallel biconnectivity algorithm remained a challenge for a while. Following several less efficient parallel algorithms [18, 49, 60], Tarjan and Vishkin [58] were the first to give an optimal, $O(\log n)$-time and $O(n + m)$-work, PRAM biconnectivity algorithm.

Follow-up works included heuristic improvements to the algorithm [15], and evaluation on the *Explicit Multi-Threading* (XMT) platform with promising results [19]. More recent papers [11, 53] pointed out that Tarjan-Vishkin algorithm is not efficient on multi-core CPU. Instead, they suggested alternative heuristic solutions, without theoretical worst-case guarantees, based on breadth-first search and disregarding the Euler tour technique. Adhering to this trend, Wadwekar and Kothapalli [61] implemented a BFS-based biconnectivity algorithm on GPU, achieving significant speedups over multi-core CPU. That is the only GPU biconnectivity algorithm we are aware of.

1.2 Experimental Setup

We run our experiments on an NVIDIA GeForce GTX 980 GPU, with 2048 CUDA cores, and an Intel Xeon X5650 CPU, with 6 physical and 12 virtual cores. The reported GPU performance ignores memory transfer times between CPU and GPU[2]. The source code, in C++ and CUDA, is available at GitHub[3].

2 The Euler Tour Technique

The Euler tour technique is a classical PRAM method for representing and performing computations on trees. For problems where an input graph is not necessarily a tree (e.g. the biconnectivity problem) the technique is usually applied to a spanning tree of the input graph.

The crux of the approach is to represent a (rooted) tree as a (singly linked) list of directed half-edges in the order of a depth-first search traversal (see Figure 1), i.e. as an Euler tour.

Note that every subtree (a subgraph induced by a node and all of its descendants) corresponds to an interval in the list. Hence many node statistics can be easily calculated as prefix sums or range queries. For example, one can assign weight 1 to each edge going down[4], and weight 0 to each edge going up, and compute the prefix sums to obtain the preorder numbering of the nodes. If the edges going up are instead assigned weights -1, the prefix sums are equal to the node levels (i.e. distances from the root).

In the PRAM model such prefix sums can be computed with a simple modification of a list-ranking algorithm, which can be implemented in optimal $O(\log n)$ time on $O(n/\log n)$ processors [2, 13]. For a practical GPU algorithm, however, we need a more careful approach (see Section 2.2).

[2]Following the usual practice, we assume our algorithms are meant to be parts of a larger GPU processing pipeline, and thus the input data is already present in the GPU memory.

[3]`https://github.com/stobis/euler-meets-cuda/`

[4]Note that a directed half-edge goes down if and only if it appears before its twin (i.e. the opposite direction half-edge) in the list.

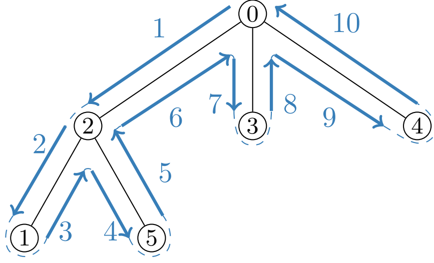


Figure 1: Example of an Euler tour

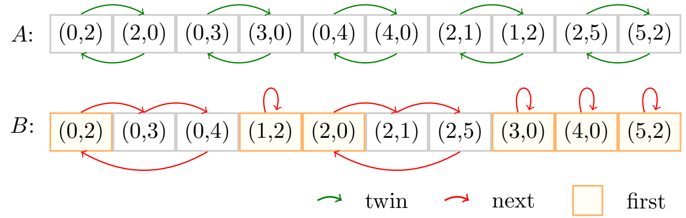


Figure 2: Intermediate DCEL-like representation of the tree from Figure 1

2.1 Constructing an Euler Tour

Since a tree is rarely already present in the Euler tour representation, an important part of the technique is constructing a tour in the first place.

The actual construction algorithm has to depend on the input tree representation. Here we present an algorithm working with a very unstructured input: an unordered collection of undirected edges, represented as pairs of node identifiers. Other common tree representations can be easily and quickly converted to this representation. The first step in the Euler tour construction is creating an intermediate DCEL-like[5] representation of the tree. We remark that, unless the tree is already in such DCEL-like representation, it is not clear if any additional structure in the input can significantly speed up the conversion, in particular by avoiding the costly sorting.

DCEL

Our intermediate goal is to represent the input n-node tree as a collection of $2 \cdot (n-1)$ directed (half-)edges, with each edge storing two pointers: *next* and *twin*. For each vertex x, all edges (x, y) starting at x are organized in a singly-linked cyclic list, so that the *next* pointer of an edge points to the next edge on that list. The *twin* pointer of each edge (x, y) points to the reverse direction edge (y, x).

Constructing DCEL

First, we create an array of directed half-edges A: For each undirected edge $\{x, y\}$ we put (x, y) and (y, x) next to each other in A. Then, we create the lexicographically sorted copy of A, which we call B. When sorting, we ensure that each element of either array keeps an up-to-date pointer to its copy in the other array. To finish the construction we observe that the *twin* pointer of an edge shall point to its neighbor in A, and the *next* pointer – to its neighbor in B, unless it is the last edge starting at a given vertex x. In order to handle such edges, we need one additional array $first$, with $first[x]$ pointing to the first edge starting at x in B. Then, whenever $B[i] = (x, y)$ and $B[i+1] = (x', y')$ for $x \neq x'$, the *next* pointer of (x, y) shall point at $first[x]$. See Figure 2.

From DCEL to an Euler Tour

Constructing an Euler tour, as a linked list, is now straightforward: The successor of an edge e in the list, $\text{succ}(e)$, is given by

$$\text{succ}(e) = \text{next}(\text{twin}(e)).$$

[5] DCEL stands for *doubly connected edge list*, which is a common data structure for planar graphs and geometric problems.

Conceptually, after traversing an edge $e = (x, y)$ and arriving at vertex x, one looks where they came from, i.e. $\text{twin}(e) = (y, x)$, and departs using the next edge, i.e. $\text{next}(\text{twin}(e))$. For example, looking at Figure 1, $\text{succ}(6) = \text{next}(\text{twin}(6)) = \text{next}(1) = 7$.

Technically, a list created in such a way is cyclic. In order to perform, say, a prefix-sum computation we need to split the list at some point. We do it at an arbitrary edge (r, y) leaving the root r of the tree. To put it differently, if we start with an unrooted tree, we choose the root by choosing the list head.

2.2 Using an Euler Tour on GPU

List-ranking is a common building block for PRAM algorithms (see [32] for an overview). In its basic variant, the list-ranking operation takes as input a list of elements, and computes for each element its *rank*, i.e. the distance from the head of the list. It can be implemented in optimal $O(\log n)$ time on $O(n/\log n)$ processors [2, 13]. Moreover, a simple adaptation of the basic algorithm allows to compute the prefix sum of arbitrary values stored in the list, without increasing the asymptotic complexity. That happens to be the same complexity as that of computing the prefix sum in an array. That is, in the theoretical PRAM model, the prefix sums in an array and in a list cost the same.

In practice, however, on GPU the scan operation (i.e. array prefix sum) is much faster then list-ranking (see e.g. [64], who report 7-8x difference). Since most Euler tour applications involve several prefix sum computations on the tour, we propose the following important optimization. After calculating an Euler tour as a list, we call a single list-ranking on the list, and we use its output to create an array of (directed) edges in the Euler tour ordering. That allows us to perform all the following prefix sum calculations on the Euler tour by using a fast scan primitive on the array. Moreover, knowing the indices of an edge and its twin, we can determine if the directed edge goes up or down. As a result we can, e.g., easily determine parents of all nodes, which we do in the *hybrid* algorithm proposed at the end of Section 4.3.

We use the *moderngpu* library [6] for fast *sort* (in DCEL construction) and *scan* primitives. Using the library throughout the implementation saves us the burden of low-level fine tuning, typically required to achieve a good performance on GPU. For list-ranking we implement the GPU-optimized Wei-JaJa algorithm [64], which performs much better than the classical pointer jumping technique.

3 First Application: Lowest Common Ancestors

For a rooted tree T and two nodes $x, y \in T$, the lowest common ancestor of x and y is the unique node furthest from the root which is an ancestor of both x and y. Equivalently, it is the only node on the unique simple path from x to y which is an ancestor of both x and y.

Given a rooted tree T and a sequence of queries $(x_1, y_1), (x_2, y_2), \ldots, (x_q, y_q) \in T \times T$, the LCA problem asks to compute for each query (x_i, y_i), $i \in \{1, 2, \ldots, q\}$, the lowest common ancestor of x_i and y_i.

3.1 Algorithms

There is a diverse collection of algorithms computing LCA with a (near-)linear preprocessing time and a constant or logarithmic query time, see e.g. [20]. Two main approaches emerge among them: (1) an approach based on a tree decomposition, e.g. [50], and (2) an approach based on the reduction to the range minimum query (RMQ) problem, e.g. [9].

The Inlabel algorithm [50] falls into the first category. It has a linear preprocessing time and a constant query time. Moreover, it was designed to be easily parallelizable. With the Euler tour technique, the preprocessing can be implemented to run in $O(\log n)$ time on $O(n/\log n)$ processors in the PRAM model. Each query can be then answered in constant time on a single

processor, i.e. a collection of q queries can be answered in $O(1)$ time on $O(q)$ processors. Besides the Euler tour technique the algorithm requires no design changes to implement it on GPU.

In order to select a single-core CPU baseline, we run a preliminary experiment with a sequential implementation of the Inlabel algorithm against a simple RMQ-based algorithm – a variant of [9], using a segment tree and without the preprocessed lookup tables for all short sequences. The RMQ-based algorithm has a faster preprocessing, by a factor of two, and the Inlabel algorithm answers queries faster, by a factor of three. When the number of queries equals the number of nodes, the two algorithms perform on par with each other. We chose the Inlabel algorithm as our single-core CPU baseline. We also implemented, using OpenMP, the parallel Inlabel algorithm as our multi-core CPU baseline (we are not aware of any publicly available multi-core LCA algorithm).

The Inlabel Algorithm

In a full binary tree the lowest common ancestor of nodes x and y can be found by examining the longest common prefix of the inorder numbers of x and y. That can be done with a constant number of bitwise operations. The Inlabel algorithm [50] generalizes that observation to general trees.

Given a rooted tree T, the algorithm assigns to each node an *inlabel* number, so that the two conditions are satisfied:

- *Path partition property.* The inlabel numbers partition T into paths going top-down, each path consisting of nodes with the same inlabel number.
- *Inorder property.* Let B be the smallest full binary tree having at least $|T|$ nodes. We identify each node of B with its inorder number. The inlabel numbers map the nodes from the input tree T to the nodes of B, not necessarily injectively, so that for every $x, y \in T$, if x is a descendant of y (in T) than inlabel(x) is a descendant of inlabel(y) (in B).

Schieber and Vishkin [50] showed how to calculate the inlabel numbers, from the preorder numbers and subtree sizes, in $O(1)$ time per node, in parallel. Next, they showed how – having calculated the inlabel numbers and two other auxiliary arrays – to answer LCA queries in constant time, on a single processor per query. We refer to the original paper [50] for a detailed description.

The Euler tour technique can be used to calculate the preorder numbers, subtree sizes, and node depths, in $O(\log n)$ time and $O(n)$ total work. The remaining part of the preprocessing runs in $O(1)$ time and $O(n)$ total work. Apart from the Euler tour technique itself, adapting the algorithm from PRAM to GPU is straightforward.

The Naïve Algorithm

Martins et al. [38] propose a naïve LCA algorithm for GPU: Each thread gets assigned a single LCA query (x, y), and it traverses the tree upwards from x and y, node by node, until the two paths meet.

Each query takes time proportional to the distance from x to y, compared to $O(1)$ time of the theoretically optimal Inlabel algorithm. However, if one expects that typical instances have short x-y paths, e.g. the trees are shallow, then the naïve algorithm benefits from its extreme simplicity and low constant factors.

In principle it is possible to implement the algorithm without any preprocessing at all. However, that would then require threads to mark visited nodes and, as a consequence, to issue scattered writes to large linear-size arrays – a thing to avoid for an efficient implementation. Fortunately, a simple and effective preprocessing lets us handle queries in a constant memory.

The preprocessing amounts to calculating the *level* of each node, i.e. the distance from the root. We adapt a standard pointer jumping technique for this task: Each node gets an *ancestor*

pointer, initially pointing to its parent, and level 0 is assigned to the root. Then, until all levels are calculated, the ancestor pointers are updated so that their lengths double, i.e. the ancestor of a node is updated to be the ancestor of its ancestor. The level of a node is calculated once its ancestor pointer points at the root or other node with an already known level. The preprocessing runs in $O(\log n)$ time and $O(n \log n)$ total work.

We remark that this is not a theoretically optimal way to compute the node levels, but our experiments show that is already fast enough never to be a bottleneck. We do however employ a simple practical optimization: We perform five jumps for each pointer in parallel, before synchronizing the threads globally, as this empirically proves to be faster than synchronizing after each parallel pointer jump.

To handle a query (x, y), a thread starts with two pointers, pointing at x and y at the beginning. If the levels of x and y are different, the pointer with the higher level is moved upwards, node by node, the number of times equal to the difference of levels, so that both pointers point at nodes at the same level. Then, both pointers are move upwards simultaneously, until they point at the same node, which is $\mathrm{LCA}(x, y)$.

3.2 Datasets

We generate synthetic datasets for our experiments. That gives us a flexibility to gradually change an isolated parameter and observe its impact on the algorithms' behavior. Besides, to the best of our knowledge, there is no established dataset for the LCA problem.

We start with a method to generate trees of a logarithmic depth. Let $[n] = \{1, 2, \dots, n\}$ be the node set. Node 1 is the root, and the parent of node i, for each $i \geq 2$, is selected uniformly at random from $\{1, \dots, i-1\}$. Trees generated that way have the expected average node depth equal to $\ln n$.

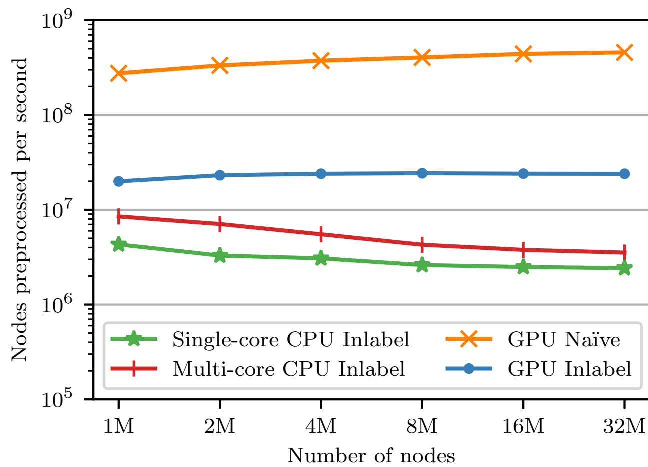
In order to be able to generate deeper trees, we introduce an additional parameter, *grasp*, which we denote by γ. Now, the parent of node i, for each $i \geq 2$, is selected uniformly at random from $\{\max(i-\gamma, 1), \dots, i-1\}$. For example, $\gamma = 1$ deterministically yields a path, and $\gamma = \infty$ recovers the initial shallow tree distribution. In general,

$$\mathbb{E}(\text{average node depth}) = \begin{cases} \ln n, & \text{if } \gamma = \infty, \\ \frac{1}{\gamma+1} n + O(1), & \text{otherwise.} \end{cases}$$

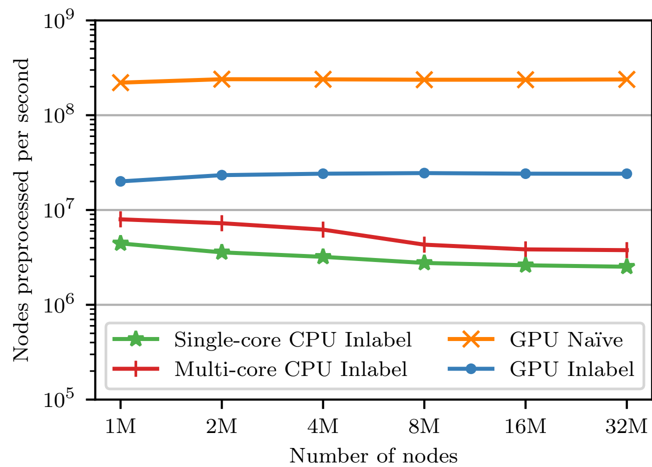
The above model lets us control the tree size and depth, but not other parameters, e.g. the degree distribution. For our final LCA experiment we generate scale-free Barabási-Albert trees [4, 42]. The parent of node i is again selected from $\{1, \dots, i-1\}$, but with probabilities proportional to the degrees, instead of the uniform distribution. Such trees have power law degree distributions, resembling real-world networks, and are very shallow on average.

At the end, we map all node identifiers through a random permutation of $[n]$, so that the tree structure is maintained but the identifiers do not leak any information. We sample queries uniformly at random from $[n] \times [n]$.

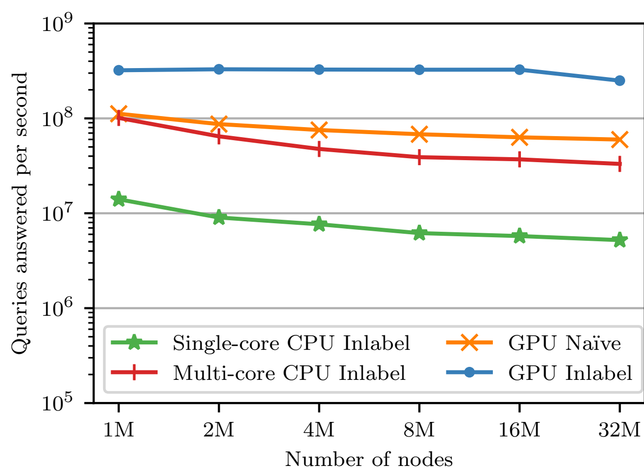
The input is given to the algorithms as an array of parents – i.e. node $P[i]$ is the parent of node i, for every i except for the root – and an array of queries. We remark that such tree description gives a head start to the naïve algorithm, while the Inlabel algorithm is essentially indifferent to the input format.



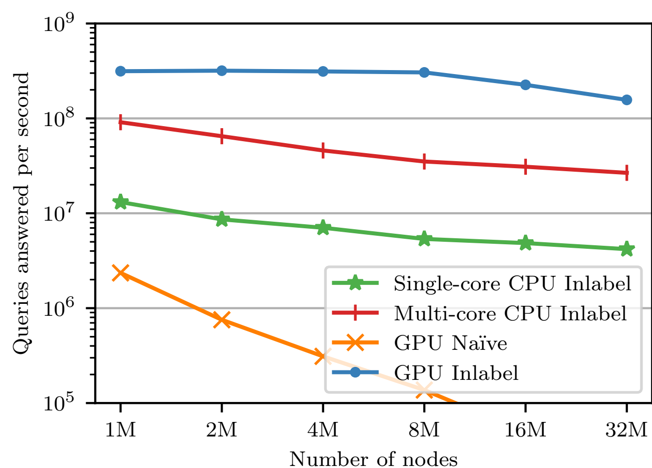
(a) Preprocessing shallow trees



(b) Preprocessing deep trees



(c) Answering queries in shallow trees



(d) Answering queries in deep trees

Figure 3: General comparison of LCA algorithms

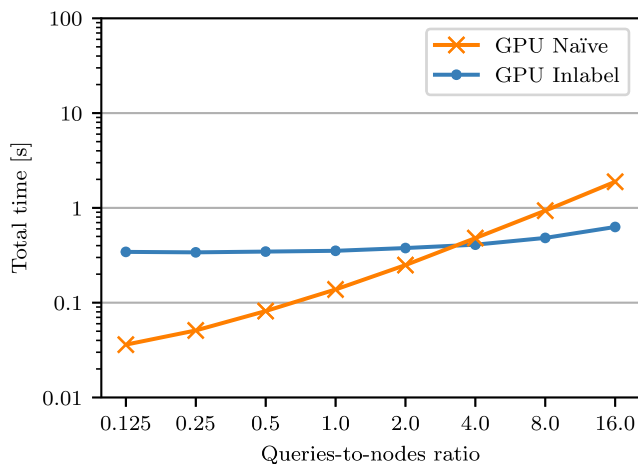


Figure 4: Combined preprocessing and query time, depending on number of queries (8M nodes, shallow)

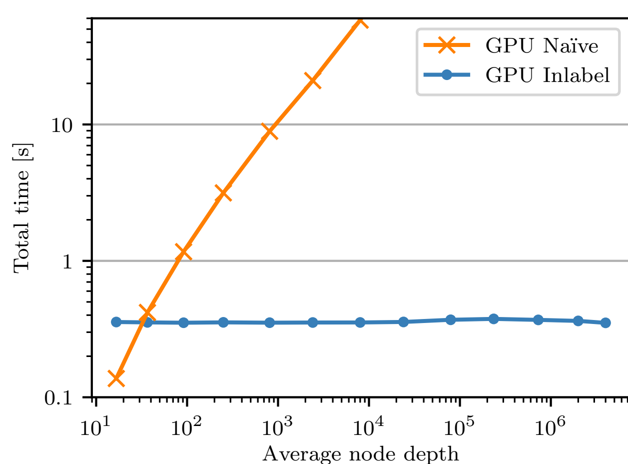


Figure 5: Time to answer 8M LCA queries in 8M-node tree, depending on tree depth (including preprocessing)

3.3 Experiments and Results

For each set of parameters (i.e. the number of nodes, number of queries, and grasp) that we considered, we generated five instances with different random seeds. Then, we run each algorithm on each instance five times. That gives 25 runs for each data point, and we report the average running times over those runs. The standard deviation, over the 25 runs, never exceeded 6% for the GPU algorithms, 13% for the single-core CPU algorithm and 18% for the multi-core CPU algorithm. The standard deviation between the five runs on the same instance was below 6% for all the algorithms.

General Comparison

In our first experiment we generated trees of size varying from one million to 32 million nodes, and fed them to all four algorithms: (1) the baseline single-core CPU implementation of the Inlabel algorithm, (2) the multi-core CPU Inlabel algorithm, (3) the supposedly practical naïve GPU algorithm, and (4) the theoretically optimal GPU implementation of the Inlabel algorithm. For each tree size we considered shallow trees ($\gamma = \infty$) and deep trees ($\gamma = 1000$). We set the number of queries to be equal to the number of nodes. The throughput of the algorithms, separately for the preprocessing and queries, is presented in Figure 3.

Let us first look at the shallow trees (Figures 3a and 3c). Both GPU algorithms offer a speedup over the CPU baselines. Unsurprisingly, the naïve algorithm has the fastest preprocessing, and the parallel Inlabel algorithm answers queries fastest. Which of the two algorithms is faster in total will depend on the ratio of the tree size to the number of queries (see Figure 4, which we analyse later).

Now, let us move our attention to the deep trees (Figures 3b and 3d). The preprocessing performance does not differ significantly compared to the shallow trees. However, the query performance of the naïve GPU algorithm degrades heavily, to the point that it even becomes slower than the single-core CPU baseline. The other three implementations, all based on the Inlabel algorithm, behave similarly as for the shallow trees.

Regardless of the tree depth, the GPU implementation of the Inlabel algorithm consistently outperforms its reference single-threaded CPU implementation, offering a speedup in the range of 4-11x for preprocessing and 22-52x for queries. The multi-threaded CPU implementation situates between the other two.

Two parameters – the number of queries relative to the number of nodes, and the tree depth – determine whether the naïve algorithm outperforms the GPU Inlabel implementation. In our next two experiments we explore the parameter regimes in which each of the two algorithms thrives.

Queries-to-Nodes Ratio

In our next experiment we focused on shallow trees (which give the most advantage to the naïve algorithm), and varied the queries-to-nodes ratio. Specifically, we fixed the number of nodes to 8 million, and varied the number of queries from one million to 128 million (thus varying the queries-to-nodes ratio from 0.125 to 16). We report the total running times, including preprocessing and answering all queries, in Figure 4. The parallel Inlabel algorithm begins to outperform the naïve algorithm at around 4-to-1 queries-to-nodes ratio. Since the throughput of both algorithms does not depend much on the number of nodes, i.e. the lines plotted in Figures 3a and 3c are roughly horizontal, we expect that the intersection point would fall in similar places also for other tree sizes.

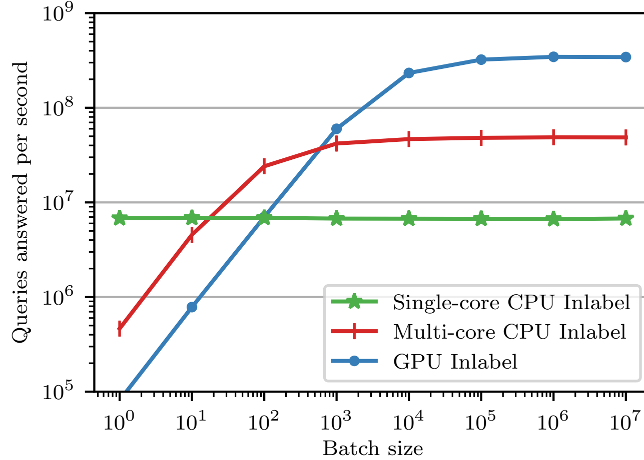


Figure 6: Benefit of answering LCA queries in parallel depending on batch size

Tree Depth

Next, we explored how the tree depth impacts the algorithms' performance. We fixed the number of nodes and the number of queries, both to 8 million, and varied the grasp parameter from 1 to 10^7, effectively generating trees with the average node depth ranging from 16 to $4 \cdot 10^6$. We report the total running times, including preprocessing and answering all queries, in Figure 5.

As we already could expect from the initial general comparison experiment, the running time of the GPU Inlabel implementation was consistent across tree depths, and equaled to about 350 milliseconds. The naïve algorithm was 2.6x faster on the least deep trees. However, already at 91 average node depth the two algorithms drew, and the naïve algorithm's performance degraded very quickly with the increasing depth.

Batch Size

The LCA algorithms which we consider in this paper can work *online*, i.e. they can preprocess a tree without knowing the queries in advance, and then they can efficiently answer queries one by one. That has a benefit of, e.g., a small latency between providing a query and getting the answer for that query, and the ability to adapt future queries based on answers to previous queries. On the other hand, in order to benefit from solving the LCA problem on a parallel machine, such as multi-core CPU or GPU, the machine has to work on many queries at once.

In our next LCA experiment we explore the scenario in which not all queries are known beforehand yet they can be grouped into batches of certain size, and each batch can be given to an LCA algorithm at once. We are interested in how the batch size impacts the performance of the Inlabel algorithm.

We fixed the number of nodes to 8 million, and grasp to infinity. We generated 10 million random queries, split them into batches, and fed them to each algorithm batch by batch. The batch size varied from 1 to 10^7. Unsurprisingly, as batch size grew, the query time on multi-core CPU and GPU decreased (see Figure 6). GPU was faster than single-core CPU with as few as 100 queries in a batch, and achieved its maximal throughput already at the batch size equal to 10 000. Multi-core CPU beat single-core already just after 10 queries in a batch, and plateaued at around 1000 queries in a batch, where GPU took the lead.

Scale-Free Trees

In our last LCA experiment we again test all four algorithms, this time on scale-free Barabási-Albert trees. Similarly to our first LCA experiment (Figure 3), the number of nodes varies from

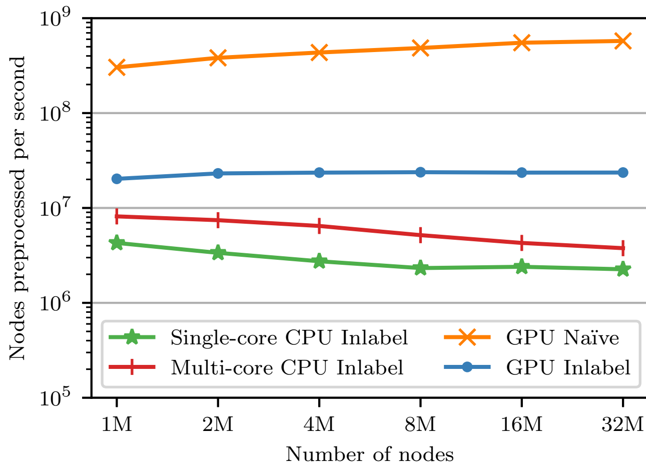


Figure 7: Preprocessing scale-free trees

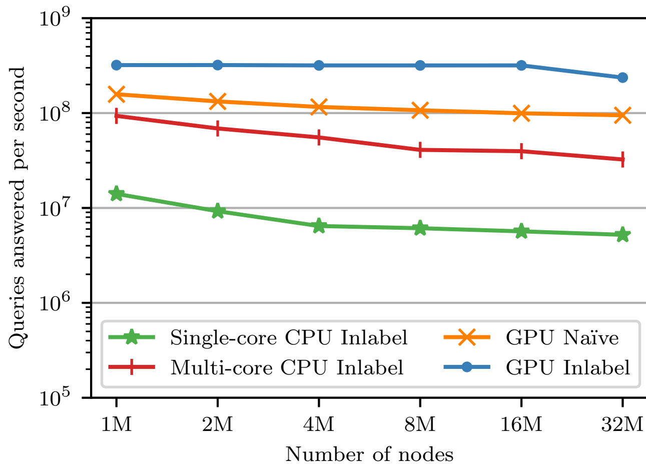


Figure 8: Answering queries in scale-free trees

one million to 32 million, the number of queries equals to the number of nodes, and we present the throughput of the algorithms separately for the preprocessing and queries, in Figures 7 and 8 respectively.

Unsurprisingly, these figures are very similar to Figures 3a and 3c for shallow trees, generated with $\gamma = \infty$. The only observable difference is the naïve GPU algorithm answering queries slightly faster, due to an even lower average node depth of the scale-free trees. The conclusions from our first LCA experiment remain valid also for the scale-free trees. This experiment confirms that the performance of the tested algorithms depends almost entirely on the tree size – and does not depend on other parameters, such as the tree shape, degree distribution, etc. – with the exception of the naïve GPU algorithm, whose query answering performance depends heavily also on the tree depth.

4 Second Application: Bridges

In a connected undirected graph, a *bridge* is an edge whose deletion makes the graph no longer connected. A *2-edge-connected component* is a maximal subgraph which does not contain a bridge. A simple method to decompose a graph into 2-edge-connected components is to find all bridges, remove them, and find connected components in the resulting graph. Closely related notions of an *articulation point* and a *2-vertex-connected component* are defined similarly for vertices.

For the sake of simplicity, in this paper we focus on the following problem: Given a graph $G = (V, E)$, determine for each edge $e \in E$ whether e is a bridge. This basic problem already captures most of the combinatorial structure related to biconnectivity. In particular, the classical bridge-finding algorithm [30] uses the *low* function, which is a central tool also for 2-edge- and 2-vertex-connected components decomposition, and which is tightly related to depth-first search trees – a major obstacle for an efficient parallelization [45].

4.1 Algorithms

First we recall the classical (sequential) bridge-finding algorithm [29,30,43]. Let us fix a spanning tree T, and let us identify the nodes with their preorder numbers. For a node $v \in T$, we define $\text{low}(v)$ to be the minimum (preorder number) of endpoints of non-tree edges whose other endpoints belong to the subtree rooted in v. The low function can be easily computed in linear

time. If T is a depth-first search tree, then a tree edge[6] $\{u, \mathrm{parent}(u)\}$ is a bridge if and only if $\mathrm{low}(u) \geq u$. Said differently, if it is possible to escape the subtree of u, it is possible to do so by going to a vertex smaller than u (in the preorder numbering), which is specifically a property of DFS trees.

Tarjan-Vishkin (TV) Algorithm

Tarjan [56] proposed how to modify the above algorithm to work with any spanning tree – thus escaping the DFS parallelization obstacle. Tarjan and Vishkin [58] showed how to implement the modified algorithm in parallel on PRAM.

In addition to the *low* function, defined as above, the algorithm also computes the *high* function, which is defined similarly but with maximum in place of minimum. For any spanning tree T, a tree edge $\{u, \mathrm{parent}(u)\}$ is a bridge if and only if at least one of $\mathrm{low}(u)$ and $\mathrm{high}(u)$ points outside of the subtree of u, i.e. outside of the interval $\big[\mathrm{preorder}(u), \mathrm{preorder}(u) + \mathrm{size}(u)\big)$, where $\mathrm{size}(u)$ denotes the size of the subtree.

The algorithm has three phases: (1) constructing a spanning tree, (2) rooting the tree and calculating required node statistics, and (3) computing the *low* and *high* functions in order to find bridges.

We use a GPU-optimized connected components algorithm by Jaiganesh and Burtscher [31], which constructs a spanning tree as a byproduct.

Using the Euler tour technique, we root the tree, calculate the preorder numbers and subtree sizes, and for each node its minimum and maximum non-tree neighbor. For the last two values we use a specialized scan implementation *segreduce* from the *moderngpu* library [6].

In order to compute the *low* and *high* functions we need to aggregate, over subtrees, the per-node minimum and maximum non-tree neighbors. That tasks boils down to solving the range minimum query (RMQ) problem, which we do using the *segment tree* data structure.

Chaitanya-Kothapalli (CK) Algorithm

The alternative CK bridge-finding method is a simple, worst-case quadratic work, heuristic algorithm. It involves multiple walks on the edges of the graph, and works particularly well for graphs with a small diameter. Implementations of the CK algorithm are state-of-the-art parallel biconnectivity algorithms for multi-core CPU [11] and GPU [61].

The algorithm consists of two phases. First, it finds a (rooted) spanning tree of the input graph. In the second phase, for each non-tree edge in parallel, it starts from its endpoints and walks up the tree up to their lowest common ancestor, marking tree edges visited along the way. A tree edge is a bridge if and only if it never get marked. For a detailed description and a proof of correctness see [11, 61].

A spanning tree algorithm for the first phase can be chosen arbitrarily, but a parallel BFS is used in most implementations [11, 61]. The choice of BFS guarantees that the spanning tree depth is at most a factor of two from the minimum, which allows to bound the work performed in the marking phase by $O(md)$, where d denotes the diameter. Moreover, BFS is a very well studied graph primitive, with highly-optimized implementations available for many parallel environments.

Our multi-core CPU implementation of the CK algorithm is based on the publicly available implementation of the runner-up algorithm [52] using OpenMP. Our GPU implementation uses our own implementation of BFS, based on [39] and using *moderngpu* primitives [6].

[6] A non-tree edge is never a bridge.

Graph	Nodes	Edges	Bridges	Diameter
kron_g500-logn16	55K	4.9M	12K	6
kron_g500-logn17	107K	10M	26K	6
kron_g500-logn18	210K	21M	54K	6
kron_g500-logn19	409K	43M	113K	7
kron_g500-logn20	795K	89M	233K	7
kron_g500-logn21	1.5M	182M	477K	7
web-wikipedia2009	1.8M	9.0M	1.4M	323
cit-Patents	3.7M	33M	1.3M	26
socfb-A-anon	3.0M	47M	3.3M	12
soc-LiveJournal1	4.8M	85M	2.2M	20
ca-hollywood-2009	1.0M	112M	23K	12
USA-road-d.E	3.5M	8.7M	2.2M	4K
USA-road-d.W	6.2M	15M	3.8M	4K
great-britain-osm	7.7M	16M	4.8M	9K
USA-road-d.CTR	14M	34M	8.5M	6K
USA-road-d.USA	23M	58M	14M	9K

Table 1: Statistics of largest connected components in graphs used in bridge-finding experiments

4.2 Datasets

We evaluated the bridge-finding algorithms on 16 graphs, listed in Table 1. They can be split into three main categories: (1) synthetic Kronecker graphs [35], which model moderately sparse networks with small diameters, (2) real-world Internet and social network graphs, with similar characteristics to Kronecker graphs, and (3) real-world road graphs, which are extremely sparse and have significantly larger diameters. The graphs are available for download from public repositories [3, 17, 36, 46], and they commonly appear in experimental papers on biconnected components, e.g. [11,61], and GPU graph algorithms in general, e.g. [63]. We preprocessed each graph to keep only its largest connected component. Table 1 summarizes basic statistics of the final test instances.

4.3 Experiments and Results

We ran each algorithm on each graph 5 times, and we report the average running times. The standard deviation was below 5% for the four algorithms discussed in Section 4.1, i.e. TV and CK algorithms on GPU, and DFS and CK on CPU. For the hybrid algorithm, proposed at the end of this section, the standard deviation was below 8.5%.

First, let us just focus on the total running times, to see which of the algorithms performs best. See Figure 9 for the performance on Kronecker graphs, and Figure 10 for real-world graphs.

Comparing the two GPU algorithms, we see that TV is faster than CK on all instances but two: the smallest Kronecker graph, and the Wikipedia graph. The difference in favor of TV is most sig-

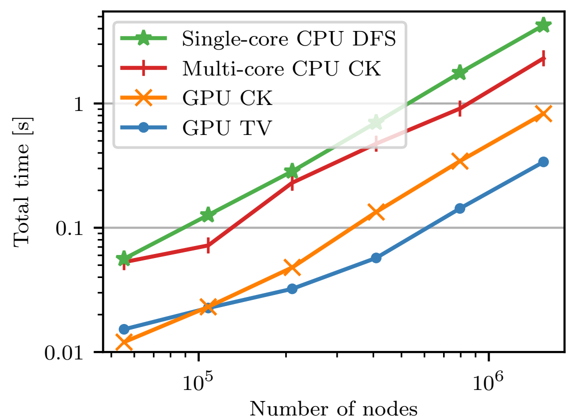


Figure 9: Comparison of bridge-finding algorithms on Kronecker graphs

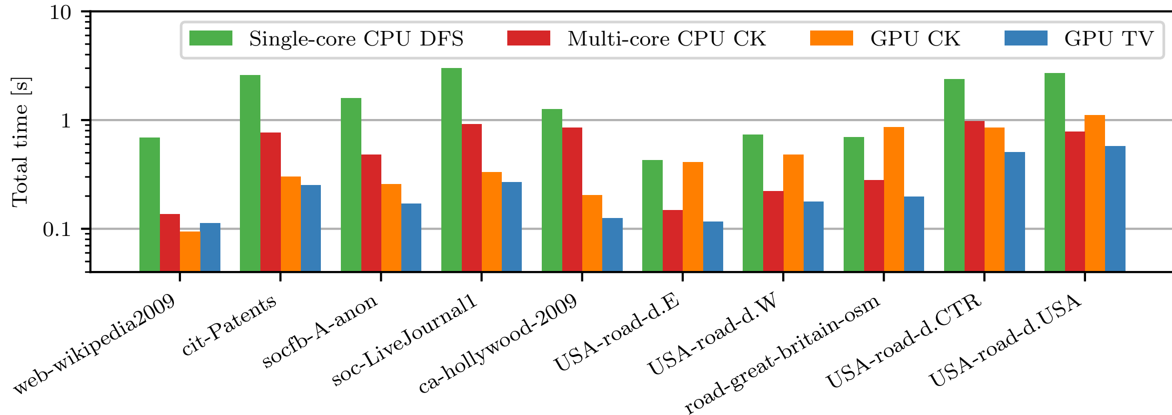


Figure 10: Comparison of bridge-finding algorithms on real-world graphs

nificant for the road graphs with large diameter, where TV is up to 4.7x faster than CK.

When compared to the CPU baselines, the TV algorithm shows 4-12x speedups over the single-core DFS implementation and 1.25-8x speedups over the multi-core CK implementation. A speedup of less than 3x over the multi-core algorithm can be observed only for the road graphs and the small Wikipedia graph.

The Hybrid Algorithm

In order to better understand the performance of the algorithms we can look at Figure 11, which presents what contributes to their running times. For the GPU CK implementation BFS always accounts for a significant portion of the running time, and becomes an apparent bottleneck when the input diameter grows. That is not surprising given the nature of parallel BFS performance, which is very sensitive to the diameter (see e.g. [39]).

The correctness of the marking phase of the CK algorithm does not depend on specific properties of breadth-first search trees. Hence, a natural attempt to speed up the algorithm is to replace BFS with a different, faster algorithm computing a spanning tree. Such tree is likely to be deeper than a BFS tree, but the hope is that the performance of the marking phase degrades only slightly, and the time saved on finding a spanning tree is not entirely lost.

We propose to compute spanning trees with the same connectivity algorithm [31] which we use in the TV implementation. It is important to note that this algorithm outputs an unrooted spanning tree, but the marking phase requires a rooted tree, i.e. each node has to know its parent. Additionally, each node also needs to know its level (distance from the root). We compute both parents and levels using the Euler tour technique.

Looking again at Figure 11, we can see that the hybrid algorithm is unlikely to outperform TV, for the following reason. Both algorithms begin with spanning tree and Euler tour computations. Then, the hybrid algorithm still has to execute the marking phase, which is likely to be no faster than the marking phase of CK, which in turn is (on most of the test instances) slower than the last remaining stage of TV.

We evaluated the hybrid algorithm experimentally – as predicted, it was often faster then CK, but it never outperformed TV. We leave it for further research whether there is a faster method to get a rooted spanning tree, without resorting to the Euler tour technique, and whether a resulting hybrid algorithm could beat TV.

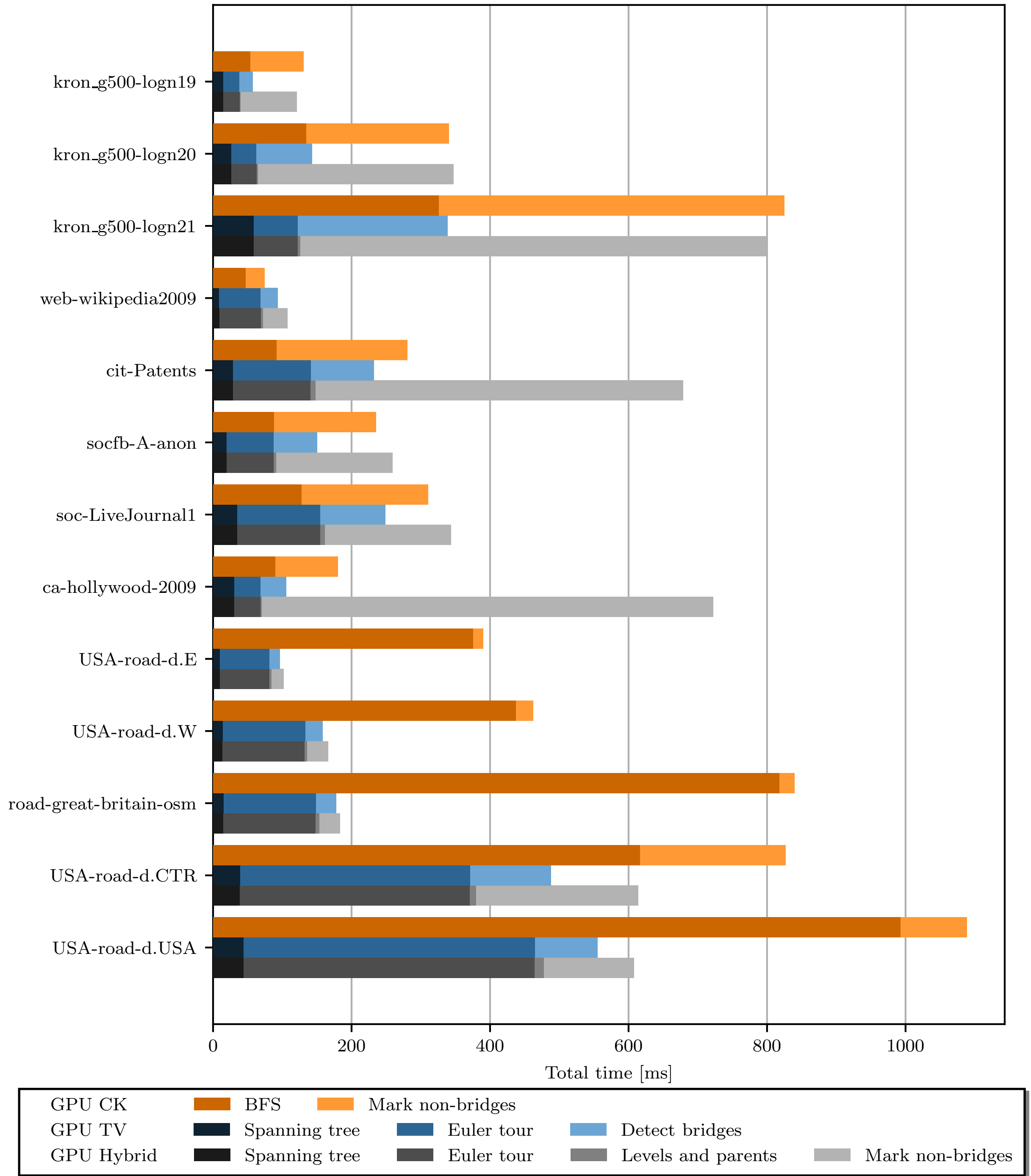


Figure 11: Running time breakdown of GPU bridge-finding algorithms

5 Conclusions

We propose how to adapt to GPU a classical building block of optimal parallel graph algorithms, the Euler tour technique. As a proof of concept we study two natural graph problems for which, to the best of our knowledge, the only previous GPU algorithms are based on simple heuristics, efficient for typical problem instances. In our experiments the Euler tour-based algorithms outperform the previous approaches unless the tree is shallow and the queries are few, in case of the LCA problem, and unless the graph is small, in case of bridge-finding.

Acknowledgments

We thank anonymous reviewers for many helpful suggestions and comments. We also thank Grzegorz Gutowski for his indispensable support in running the experiments.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 253–265, 1973. doi:10.1145/800125.804056.

[2] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In *VLSI Algorithms and Architectures*, pages 81–90, 1988. doi:10.1007/BFb0040376.

[3] David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering. Proceedings of the 10th DIMACS implementation challenge workshop, 2012. 03 2013. doi:10.1090/conm/588.

[4] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

[5] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on CUDA. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 544–555. IEEE, 2011. doi:10.1109/IPDPS.2011.59.

[6] Sean Baxter. moderngpu 2.0, 2016. URL: https://github.com/moderngpu/moderngpu.

[7] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. Massively parallel computation via remote memory access. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 59–68, 2019. doi:10.1145/3323165.3323208.

[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 235–248, 2017. doi:10.1145/3018743.3018756.

[9] Michael A. Bender and Martín Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, pages 88–94, 2000. doi:10.1007/10719839_9.

[10] Ulrik Brandes and Dorothea Wagner. *Analysis and Visualization of Social Networks*, pages 321–340. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-642-18638-7_15.

[11] Meher Chaitanya and Kishore Kothapalli. Efficient multicore algorithms for identifying biconnected components. *International Journal of Networking and Computing*, 6(1):87–106, 2016. doi:10.15803/ijnc.6.1_87.

[12] Luigi Cinque, Sergio De Agostino, and Luca Lombardi. Scalability and communication in parallel low-complexity lossless compression. *Mathematics in Computer Science*, 3(4):391–406, 2010. doi:10.1007/s11786-010-0034-5.

[13] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

[14] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3(1-4):329–346, 1988. doi:10.1007/bf01762121.

[15] Guojing Cong and David A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005. doi:10.1109/IPDPS.2005.100.

[16] Maxime Crochemore and Wojciech Rytter. Parallel construction of minimal suffix and factor automata. *Information Processing Letters*, 35(3):121–128, 1990. doi:10.1016/0020-0190(90)90060-b.

[17] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem, Proceedings of a DIMACS Workshop, 2006*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 2009. doi:10.1090/dimacs/074.

[18] D. M. Eckstein. BFS and biconnectivity. Technical Report 79-11, Deptartment of Computer Science, Iowa State University of Science and Technology, 1979.

[19] James A. Edwards and Uzi Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 103–114, 2012. doi:10.1145/2141702.2141714.

[20] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006*, volume 4009, pages 36–48, 2006. doi:10.1007/11780441_5.

[21] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 434–443. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320229.

[22] Hillel Gazit. A deterministic parallel algorithm for planar graphs isomorphism. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 723–732, 1991. doi:10.1109/SFCS.1991.185440.

[23] Alan M. Gibbons, Amos Israeli, and Wojciech Rytter. Parallel O(log n) time edge-colouring of trees and Halin graphs. *Information Processing Letters*, 27(1):43–51, 1988. doi:10.1016/0020-0190(88)90080-4.

[24] Michael T. Goodrich, Mujtaba R. Ghouse, and J. Bright. Sweep methods for parallel computational geometry. *Algorithmica*, 15(2):126–153, 1996. doi:10.1007/BF01941685.

[25] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

[26] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High Performance Computing – HiPC 2007*, pages 197–208, 2007. doi:10.1007/978-3-540-77220-0_21.

[27] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, India, 2009.

[28] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 519–527, 1995. doi:10.1145/225058.225269.

[29] John Hopcroft and Robert Tarjan. Efficient algorithms for graph manipulation. Technical Report 207, Computer Science Department, Stanford University, 1971.

[30] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.

[31] Jayadharini Jaiganesh and Martin Burtscher. A high-performance connected components implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 92–104, 2018. doi:10.1145/3208040.3208041.

[32] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[33] Goossen Kant. *Algorithms for drawing planar graphs*. PhD thesis, 1993. URL: https://dspace.library.uu.nl/bitstream/handle/1874/842/full.pdf.

[34] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/582034.582089.

[35] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042, 2010. URL: https://dl.acm.org/citation.cfm?id=1756039.

[36] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[37] Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang. 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*, PMAM'17, pages 48–57, 2017. doi:10.1145/3026937.3026941.

[38] Wellington Santos Martins, Thiago Fernando Rangel, Divino Cesar S. Lucas, Elias B. Ferreira, and Edson Cáceres. Phylogenetic distance computation using CUDA. In *Advances in Bioinformatics and Computational Biology – 7th Brazilian Symposium on Bioinformatics, BSB 2012*, volume 7409, pages 168–178, 2012. doi:10.1007/978-3-642-31927-3_15.

[39] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012. doi:10.1145/2145816.2145832.

[40] Paulius Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'04*, volume 3, pages 1359–1365, 2004.

[41] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994. doi:10.1016/0304-3975(94)90159-7.

[42] Tamás F. Móri. The maximum degree of the Barabasi-Albert random tree. *Comb. Probab. Comput.*, 14(3):339–348, 2005. doi:10.1017/S0963548304006133.

[43] Keith Paton. An algorithm for the blocks and cutnodes of a graph. *Communications of the ACM*, 14(7):468–475, 1971. doi:10.1145/362619.362628.

[44] Vijaya Ramachandran and John Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49(3):517–561, 1994. doi:10.1016/s0022-0000(05)80070-4.

[45] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.

[46] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.

[47] Wojciech Rytter. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science*, 47:315–321, 1986. doi:10.1016/0304-3975(86)90155-6.

[48] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Incremental algorithms for closeness centrality. In *Proceedings of the 2013 IEEE International Conference on Big Data*, pages 487–492, 2013. doi:10.1109/BigData.2013.6691611.

[49] Carla Savage and Joseph JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981. doi:10.1137/0210051.

[50] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988. doi:10.1137/0217079.

[51] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, 2007. doi:10.2312/EGGH/EGGH07/097-106.

[52] George M. Slota and Kamesh Madduri. BCC: BFS and coloring biconnected components algorithms, 2014. URL: https://github.com/HPCGraphAnalysis/bicc/tree/master/bcc-hipc14.

[53] George M. Slota and Kamesh Madduri. Simple parallel biconnectivity algorithms for multicore platforms. In *21st International Conference on High Performance Computing, HiPC 2014*, pages 1–10. IEEE Computer Society, 2014. doi:10.1109/HiPC.2014.7116914.

[54] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, pages 1–8. IEEE, 2010. doi:10.1109/IPDPSW.2010.5470817.

[55] Jyothish Soman, Kiran Kumar Matam, Kishore Kothapalli, and P. J. Narayanan. Efficient discrete range searching primitives on the GPU with applications. In *2010 International Conference on High Performance Computing, HiPC 2010*, pages 1–10. IEEE Computer Society, 2010. doi:10.1109/HIPC.2010.5713188.

[56] R. Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974. doi:10.1016/0020-0190(74)90003-9.

[57] Robert Endre Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Math. Program.*, 77:169–177, 1997. doi:10.1007/BF02614369.

[58] Robert Endre Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985. doi:10.1137/0214061.

[59] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *In Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, 2001. doi:10.1145/378580.378581.

[60] Yung H. Tsin and Francis Y. Chin. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM Journal on Computing*, 13(3):580–599, 1984. doi:10.1137/0213036.

[61] Mihir Wadwekar and Kishore Kothapalli. A fast GPU algorithm for biconnected components. In *2017 Tenth International Conference on Contemporary Computing (IC3)*, pages 1–6, 2017. doi:10.1109/ic3.2017.8284293.

[62] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP '16, pages 1–8, 2016. doi:10.1145/2915516.2915521.

[63] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and et al. Gunrock. *ACM Transactions on Parallel Computing*, 4(1):1–49, 2017. doi:10.1145/3108140.

[64] Zheng Wei and Joseph JáJá. Optimization of linked list prefix computations on multithreaded GPUs using CUDA. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010. doi:10.1109/ipdps.2010.5470455.

[65] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014. doi:10.14778/2733085.2733089.

[66] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1543–1552, 2014. doi:10.1109/TPDS.2013.111.