Parametric Multi-Step Scheme for GPU-Accelerated Graph Decomposition into Strongly Connected Components *

Stefano Aldegheri[1], Jiří Barnat[2], Nicola Bombieri[1], Federico Busato[1], and Milan Češka[3]

[1] Department of Computer Science, University of Verona, Italy
[2] Faculty of Informatics, Masaryk University, Czech Republic
[3] Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract. The problem of decomposing a directed graph into strongly connected components (SCCs) is a fundamental graph problem that is inherently present in many scientific and commercial applications. Clearly, there is a strong need for good high-performance, e.g., GPU-accelerated, algorithms to solve it. Unfortunately, among existing GPU-enabled algorithms to solve the problem, there is none that can be considered the best on every graph, disregarding the graph characteristics. Indeed, the choice of the right and most appropriate algorithm to be used is often left to inexperienced users. In this paper, we introduce a novel parametric multi-step scheme to evaluate existing GPU-accelerated algorithms for SCC decomposition in order to alleviate the burden of the choice and to help the user to identify which combination of existing techniques for SCC decomposition would fit an expected use case the most. We support our scheme with an extensive experimental evaluation that dissects correlations between the internal structure of GPU-based algorithms and their performance on various classes of graphs. The measurements confirm that there is no algorithm that would beat all other algorithms in the decomposition on all of the classes of graphs. Our contribution thus represents an important step towards an ultimate solution of automatically adjusted scheme for the GPU-accelerated SCC decomposition.

1 Introduction

Fundamental graph algorithms such as breadth first search, spanning tree construction, shortest paths, etc., are building blocks to many applications. Sequential implementations of these algorithms become impractical in those application domains where large graphs need to be processed. As a result, parallel algorithms for the processing of large graphs have been devised to efficiently use compute clusters and multi-core architectures. The transformation of a sequential algorithm into a scalable parallel algorithm, however, is not an easy task. Typically, the best sequential algorithm is not necessarily the best parallel algorithm from the practical point of view. This is especially the case of massively parallel graphics processing units (GPUs). These devices contain several hundreds of arithmetic units and can be harnessed to provide tremendous acceleration

* This work has been supported by the IT4Innovations Excellence in Science project No. LQ1602, the BUT FIT project FIT-S-14-2486, and the Czech Science Foundation grants No. GA16-24707Y and No. GA15-08772S.

for many computation intensive scientific applications. The key to effective utilization of GPUs for scientific computing is the design and implementation of data-parallel algorithms that can scale to hundreds of tightly coupled processing units following a single instruction multiple thread (SIMT) model.

In this paper we focus on the problem of decomposing a directed graph into its strongly connected components (*SCC decomposition*). This problem has many applications leading to very large graphs, including for example web analysis [16], which require high performance processing.

Parallelization of the SCC decomposition is a particularly difficult problem. The reason is that the optimal (i.e., linear) sequential algorithm by Tarjan [21] strongly relies on the depth-first search which is difficult to be computed in parallel. In our previous work [2] we have shown how selected nonlinear parallel SCC decomposition algorithms, namely the Forward-Backward (FB) algorithm [9, 17], the Coloring algorithm [19] and the OBF algorithm [3], can be modified in order to be accelerated on a vector processing SIMT architecture. In particular, we have decomposed the algorithms into primitive data-parallel graph operations and reformulated the recursion present in the algorithms by means of iterative procedures. This approach has been recently improved by warp-wise and block-wise task allocation for primitive graph operations [15, 8]. The authors of [8] have further proposed a SIMT parallelisation of multi-step algorithms by [12, 20] extending the FB algorithm and combining it with the Coloring algorithm.

This paper presents a new parametric multi-step scheme that allows us to compactly define a set of algorithms for SCC graph decomposition as well as a type of the parallelisation for individual graph operations. The scheme covers the existing algorithms and techniques mentioned above, but also introduces several new variants of the multistep algorithm. We use the scheme to carry out an extensive experimental evaluation that helps us to dissect the performance of the individual parametrisation on various classes of graphs. Our results indicate that there is no single algorithm that would outperform other algorithms on all type of graphs. Moreover, the results show that there is a nontrivial correlation between the parameterisation and the performance.

Based on the evaluation we identify, for each type of graphs, the key parameters of the scheme that significantly affect the performance and relate such behaviour to the structural properties of the graph. Such analysis is essential for designing an adaptive scheme that would either automatically select an adequate parametrisation based on a priori knowledge of the graph structure or automatically switch to a more viable parametrisation during the decomposition process. The automatic tuning of parameters is part of our current and future work.

2 Parallel Algorithms for SSC Decomposition

In this section we briefly present existing techniques and algorithms for parallel SCC decomposition that form basic building blocks for the parametric scheme.

2.1 Parallel graph algorithms for GPUs

In order to design scalable parallel graph algorithms that can effectively utilise modern GPUs, one has to consider key features of the underlying architecture

and to employ suitable data structures. Typical GPUs consist of multiple Stream Multiprocessors (SM) with each SM following the SIMT model. This approach establishes a hierarchy of threads arranged into blocks that are assigned for parallel execution on SMs. Threads are hardwired into groups of 32 called warps, which form a basic scheduling unit and execute instructions in a lock-step manner. A sufficient number of threads has to be dispatched to hide the memory access latency and maximise the utilisation. Memory requests exhibiting spatial locality are coalesced to improve the performance. A typical GPU program consists of a CPU host code that calls GPU kernels executing the same scalar sequential program in many independent data-parallel threads.

Data structures encoding the graph have to allow independent thread-local data processing and coalesced access. The adjacency list representation is typically encoded as two one-dimensional arrays [10]. One array keeps the target vertices of all the edges. The second array keeps an index to the first array for every vertex. The index points to the position of the first edge emanating from the corresponding vertex. Other data associated to a vertex are organised in vectors as well. In [2, 15, 8], techniques for improving memory consumption and access pattern for SCC decomposition algorithms have been proposed.

The core procedure of every graph algorithm is the graph traversal. The SCC decomposition algorithms build on several types of the traversal as explained in the next section. Parallelisation of this procedure fundamentally affects the overall performance of the decomposition. There exist several approaches [10, 11, 18, 5] that differ in the granularity of the task allocation (thread-per-vertex vs. warp-per-vertex vs. block-per-vertex) and in the number of vertices/edges processed during a single kernel (linear vs. quadratic parallelisation). In the context of the SCC decomposition the performance of these approaches significantly depends on the structure of the graphs and the type of the traversal. The parametric scheme presented in Section 3 captures various parallelisation strategies.

2.2 Forward-Backward algorithm

The Forward-Backward (FB) algorithm [9] represents the fundamental algorithm for parallel SCC decomposition. It is listed as Algorithm 1 and proceeds as follows. A vertex called *pivot* is selected and the strongly connected component the pivot belongs to is computed as the intersection of the forward and backward *closure* of the pivot. Computation of the closures divides the graph into four subgraphs that are all SCC-closed. These subgraphs are 1) the strongly connected component with the pivot, 2) the subgraph given by vertices in the forward closure 3) the subgraph given by vertices in the backward closure , and 4) the subgraph given by the remaining vertices. The later three subgraphs form independent instances of the same problem, and therefore, they are recursively processed in parallel. The time complexity of the FB algorithm is $\mathcal{O}(n \cdot (m+n))$ since it performs $\mathcal{O}(m+n)$ work to detect a single strongly connected component.

Practical performance of the algorithm may be further improved by performing elimination of leading and terminal trivial strongly connected components – the so-called *trimming* [17]. The Trimming procedure builds upon a topological sort elimination. A vertex cannot be part of a non-trivial strongly connected component if its in-degree (out-degree) is zero. Therefore, such a vertex can be

safely removed from the graph as a trivial SCC, before the pivot vertex is selected. The elimination can be iteratively repeated until no more vertices with zero in-degree (out-degree) exist.

In [2] we designed a GPU-acceleration of the FB algorithm that provides a good performance and scalability on regular graphs. In [15] the acceleration is improved by the linear parallelisation of the graph traversal and by a better pivot selection, which result in a performance gain including also a good performance on less regular graphs. The main limitation of the FB algorithm is that it performs $\mathcal{O}(m+n)$ work to detect a single SCC. This mitigates the benefits of the GPU-acceleration if the graph contains many small but non-trivial components.

Algorithm 1: FB

```
1  Procedure FB(V)
2  begin
3      pivot ← PivotSelection(V)
4      F ← FWD-Reach(pivot, V)
5      B ← BWD-Reach(pivot, V)
6      F ∩ B is SCC
7      in parallel do
8          FB(F \ B)
9          FB(B \ F)
10         FB(V \ (F ∪ B))
```

Algorithm 2: Coloring

```
1  Procedure Coloring(V)
2  begin
3      (maxColor, V_k) ←
       FWD-MaxColor(V)
4      for k ∈ maxColor in parallel do
5          B_k ← BWD(k, V_k)
6          B_k is SCC
7          if (V_k \ B_k ≠ ∅) then
8              Coloring(V_k \ B_k)
```

2.3 Coloring algorithm

The Coloring algorithm [19] is capable of detecting many strongly connected components in a single recursion step, however, for the price of an $\mathcal{O}(n \cdot (m+n))$ procedure. Therefore, the time complexity of the algorithm is $\mathcal{O}((l+1) \cdot n \cdot (m+n))$ where l is the longest path in the component graph.

The pseudo-code of the algorithm is listed as Algorithm 2. It propagates unique and totally ordered identifiers (colors) associated with vertices. Initially, each vertex keeps its own color. The colors are iteratively propagated along edges of the graph (line 3) so that each vertex keeps only the maximum color among the initial color and colors that have been propagated into it (maximal preceding color). After a fixpoint is reached (no color update is possible), the colors associated with vertices partition the graph into multiple SCC-closed subgraphs V_k. All vertices of a subgraph are reachable from the vertex v whose color is associated with the subgraph. Therefore, the backward closure of v restricted to the subgraph forms a SCC component that is removed from the graph before the next recursion step. Propagation procedure is rather expensive if there are multiple large components which limits the overall performance [2].

2.4 Other algorithms

Both the presented algorithms typically show limited performance and poor scalability when applied to large real-world graph instances with many nontrivial components and a high diameter. Fundamental properties of these graphs have been consider to propose a series of extensions of the FB algorithm [12] and a

multistep algorithm [20] that adequately combines the FB and Coloring algorithms. These two, originally multicore, alorithms have been recently redesigned to allow data-parallel processing [8], which led to the fastest GPU-accelerated SCC decomposition.

Barnat et al. [3] introduced the OBF algorithm that aims at decomposing the graph in more than three SCC-closed subgraphs within a single recursion step. However, unlike the Coloring algorithm, the price of the OBF procedure is $\mathcal{O}(m+n)$. Despite the better asymptotical complexity, our previous work [2] and also our more recent attempts indicate that effective data-parallelisation of the OBF algorithm is a very hard problem and the approaches based on the multistep algorithm performs generally better on SIMT-based architectures.

Very recently a multi-core version of the Tarjan algorithm based on parallelisation of depth-first search [4] has been proposed. It preserves the liner complexity of SCC decomposition and on a variety of graph instances it outperforms previous multi-core solutions. However, on real-word graphs it considerably lags behind the approaches by [12, 20] and the proposed parallelisation is principally not suitable for SIMT architectures.

3 Multi-step Parametric Scheme for SCC Decomposition

This section introduces a new multi-step scheme for SCC decomposition, which consists of two levels of parametrization. The first allows setting the individual steps of the algorithm, while the second allows defining the parallelisation strategy for the graph traversal.

3.1 Parametric multi-step algorithm

The multi-step algorithm consists of 3 steps: 1) I_t iterations of the Trimming procedure that identifies *trivial components* of the graph (see Section 2.2), 2) I_f iterations of the FB algorithm that aims at identifying *big components*, and 3) the Coloring algorithm that decomposes the rest of the graph. The algorithm parametrisation determines the values of I_t and I_f. Algorithm 3 depicts the host code for the GPU-accelerated version of the algorithm.

Algorithm 3: Parametric Multi-step

```
   Input  : G = (V, E), parameters I_t and I_f
   Output: SCC decomposition of G
1  for i = 1; i ≤ I_t ∧ scc ≠ V; i = i + 1 do
2      oneStepTrimming(G, scc)
3  for i = 1; i ≤ I_f ∧ V ≠ scc; i = i + 1 do
4      PivotSelection(G, pivots, ranges)
5      FWD-Reach(G, pivots, ranges, visited.f)
6      BWD-Reach(G, pivots, ranges, visited.b)
7      Update(scc, range, visited)
8  while terminate = false do
9      FWD-MaxColor(G, ranges, colors)
10     BWD-Reach(G, ranges, colors, visited.b)
11     Update(range, visited.b, colors, scc)
```

In the first step (lines 1-2), the kernel oneStepTrimming implements a single iteration of the trimming procedure. It identifies and eliminates vertices of G that form trivial SCCs. It stores the eliminated vertices in the array *scc*. Note that the proposed scheme does not perform the trimming procedure in the

later steps of the algorithm, i.e., within every FB iteration as in [2], since the Coloring algorithm handles the remaining trivial components more efficiently.

In the second step (lines 3-7), the algorithm selects a single pivot from the remaining (i.e., not eliminated) part of the graph, it computes the forward and backward closure for such a vertex, and it marks the four subgraphs (see Section 2.2) by using the Update kernel. Then, through further iterations of the second step, the algorithm selects multiple pivots and computes multiple closures restricted to the individual subgraphs. The array *ranges* is used to maintain the identification of the subgraphs, while the arrays *visited* indicate the vertices visited during the closure computations. The array *scc* is updated at every iteration to store all vertices that has been already identified in a SCC. For the pivot selection over multiple subgraphs, the algorithm implements the approach proposed in [8] extended to apply the heuristics defined in [20] to favour vertices with a high in-degree and out-degree. The FWD-Reach and BWD-Reach kernels implement parallel BFS visits of the graph, which have been adequately modified for providing reachability results.

The last step implements the coloring algorithm, which is iteratively applied to decompose the remaining subgraphs. The max color is propagated to the successor non-eliminated vertices, and stored in the array *colors* (line 9). The parametric BWD-Reach kernel implements the backward closure to identify a single component for each subgraph. Finally, the updating kernel partitions each subgraph into multiple subgraphs based on the max colors and updates the ranges accordingly for the next iteration.

Note that in our implementation the data associated to vertices in the form of the aforementioned arrays are merged and stored in two 32-bit arrays.

3.2 Parallelisation strategy for graph traversal

Another dimension of parametrisation relates to the way reachability procedures are implemented within the FB and Coloring parts of the Algorithm 3 (lines 5, 6, and 10 respectively).

Recall that when computing the reachability relation (closure), the longest path along which the algorithm has to traverse is given by the diameter of the graph. Assuming that the closure computation consists of multiple kernel calls, where each kernel call shortens this distance by at least one, we immediately have that the diameter of the graph also gives the bound on the number of kernel calls needed. However, there are multiple strategies how to implement such a single kernel call. If the kernel call is guaranteed to shorten the distance only by one, but its complexity itself is linear (e.g. it inspects all vertices/edges), we obtain an overall procedure that computes the closure in a quadratic amount of work in the worst case with respect to the size of the graph.

Alternatively, we may employ a strategy that mimics the serial graph traversal procedure and uses queue of vertices to be processed as the uderlying data structure (the so called frontier queue). In such the case, the complexity of the kernel call is proportional to the amount of vertices processed and the overall complexity of the procedure remains linear. And indeed, when dealing with large graphs, it has been shown that this works the best among various GPU-oriented

implementations [5]. On the other hand, the overhead introduced by the maintanance of the frontier queue may render the linear solution inefficient when applied to compute the closure operation on subgraphs with small diameter.

In our parametric scheme we, therefore, allow to specify which strategy should be used to compute the closures in individual phases. In particular, we support the three following options.

1. *Quadratic parallelisation (Q).* The closure computation is based on the quadratic parallel breadth-first search as proposed in [10]. It implements the simplest static workload partitioning and vertex-per-thread mapping, thus involving the smallest runtime computation overhead. This strategy works the best for large graphs with regular structure and small diameter.
2. *Quadratic parallelisation with Virtual Warps (Q_{VW}).* In this strategy we also employ the quadratic parallel breadth-first search, however, the workload partitioning and mapping rely on the *virtual warps* as proposed in [11]. This modification allows for almost even workload assigned to individual threads, which after all results in reduced branching divergence – an aspect very crucial for the performance of GPU algorithm. Virtual warps also allow improved coalescing of memory accesses since more threads of a virtual warp access to adjacent addresses in the global memory. This strategy is supposed to work the best for graphs with uneven edge distribution.
3. *Linear parallelisation (L).* This strategy is our own implementation of the linear closure procedure as proposed in [5]. It provides a highly tunable solution that allows efficient handling of very irregular graphs with the overhead of queue maintenance and dynamic load balancing at the runtime. This strategy should work the best for graphs with large diameter and nonuniform edge distribution.

Since the TRIMMING step is typically performed only through a couple of iterations, the strategy used in the TRIMMING kernel rely on a very light thread-per-vertex allocation and the quadratic parallelisation. The overhead of the linear or even more complex approach in this step would never pay off. The very same strategy has also been used for the implementation of the maximal color propagation in the COLORING phase (line 9 of Algorithm 3).

4 Experimental results

The experimental results have been run on a dataset of 17 graphs, which have been collected to represent very different structure of the graphs. The dataset covers both synthetic and real-world graphs from different sources and contexts such as social networks, road networks, and recursive graph models. The real-world graphs have been selected from Stanford Network Analysis Platform (SNAP) [14], Koblenz Network Collection [13], and University of Florida Sparse Matrix Collection [6], while the random and R-MAT graphs have been generated by using the GTGraph tool [1].

Table 1 summarizes the graph features in terms of number of vertices (in million), edges (in million), average degree, the percentage of vertices with out-degree equal to zero ($d(v) = 0$), out-degree standard deviation, average diameter (over 100 BFS from random sources), number of SCCs, percentage of vertices in the largest SCC, and the percentage of vertices in SCCs with size equal to one.

Graph Name	Vertices	Edges	Avg. Degree	N. of d(v) = 0	Std. Deviation	Avg. Diameter	N. of SCCs	Largest SCC	Trivial SCCs
amazon-2008[14]	0.7M	5.2M	7.0	12.0%	3.9	25.7	90,660	85%	12%
LiveJournal[14]	4.8M	69.0M	14.2	11.1%	36.1	12.6	971,232	79%	20%
Flickr[13]	2.3M	33.1M	14.4	32.3%	87.7	8.0	277,277	70%	19%
R-MAT[1]	10.0M	120.0M	12.0	20.2%	22.3	7.8	2,083,372	79%	21%
cit-Patents[14]	3.8M	16.5M	4.4	44.6%	7.8	4.2	3,774,768	0%	100%
Random[1]	10.0M	120.0M	12.0	0.0%	3.5	9.0	125	100%	0%
Pokec[14]	1.6M	30.6M	18.8	12.4%	32.1	9.9	325,892	80%	20%
Language[13]	0.4M	1.2M	3.0	0.0%	20.7	33.6	2,456	99%	1%
Baidu[13]	23.9M	58.3M	8.3	22.7%	23.2	12.8	1,503,003	28%	69%
Pre2[13]	0.7M	6.0M	9.0	0.0%	22.1	60.7	391	100%	0%
CA-road[14]	23.9M	5.5M	2.8	0.3%	1.0	655.9	2,638	100%	0%
web-Berkstan[13]	0.9M	7.6M	2.8	0.7%	16.4	465.6	109,409	49%	15%
SSCA8[1]	8.4M	99.0M	11.8	0.2%	4.4	1,535.9	55,900	97%	0%
trec-w10g[13]	1.6M	8.0M	5.0	4.4%	72.0	54.8	531,539	29%	31%
Fullchip[6]	3.0M	26.6M	8.9	0.0%	23.1	37.2	35	100%	0%
USA-road[7]	23.9M	58.3M	2.4	0.0%	0.9	6,277.0	1	100%	0%
Wiki-Talk[14]	18.3M	127.3M	9.4	93.8%	80.0	0.4	14,459,546	21%	79%

TABLE 1: Characteristics of the graph dataset.

The table underlines, for instance, that road networks, such as *CA-road* and *USA-road*, present in general a single SCC, a low average degree, and a low number of vertices with $d(v) = 0$. In contrast, social networks (*LiveJournal* and *Flickr*) and the R-MAT model show small-world network properties, which imply one large SCC and a high number of single-vertex SCCs.

We run the experiments on a linux system (Ubuntu 14.04) with a NVIDIA Kepler Tesla K40 GPU device with 12 GB of memory, CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, and gcc host compiler v. 4.8.4.

We compared three implementations: a sequential version that implements the Tarjan algorithm [21], which is considered the most efficient sequential algorithm. The data-parallel GPU implementation by Devshatwar et al. [8], the fastest GPU solution at the state of the art, and the proposed approach. Table 2 reports the results in terms of runtime (milliseconds) and performance (million of edges per seconds - MTEPS). The results of the proposed implementation are the best we obtained through the parameter configuration, as explained in the following. All the reported values are the average of ten runs. The results show that the application throughput (MTEPS) of the parallel implementations is directly related to the size and the average diameter of graphs. For instance, *cit-Patents* graph shows a high value of MTEPS due to a low average diameter and a regular degree distribution that allow a high GPU utilisation. On the other hand, the performance of the sequential version depends on the number of vertices and edges of the graphs.

Table 3 presents the configuration of the proposed parametric approach that leads to the best performance and compares such performance to those provided by the "static" solution of Devshatwar et al. The configurations are expressed in terms of which strategy is used in the FB and in the coloring step, i.e., linear (L), static quadratic (Q), and quadratic with virtual warps (Q_{VW}), and the number of iterations of the trimming and FB steps. Notations Q/L or Q_{VW}/L indicate that the two algorithms provide similar performance.

Graph Name	Sequential SCC		Devshatwar et al. [8]		Proposed implementation	
	Time	MTEPS	Time	MTEPS	Time	MTEPS
amazon-2008	162	32	16	325	17	305
LiveJournal	2,575	26	86	802	87	793
Flickr	821	40	54	611	54	611
R-MAT	9,182	13	193	621	192	625
cit-Patents	536	31	16	1,031	16	1,031
Random	10,619	11	231	519	218	550
Pokec	1,344	23	42	729	33	927
Language	75	16	29	41	22	55
Baidu	582	100	70	832	50	1,166
Pre2	127	47	30	200	19	316
CA-road	223	25	166	33	79	70
web-Berkstan	94	81	1,754	4	717	11
SSCA8	4,237	23.4	1,174	84	465	213
trec-w10g	147	54	12,508	1	2,218	4
Fullchip	547	49	506	53	72	369
USA-road	2,191	27	7,041	8	669	87
Wiki-Talk	5,835	22	18,907	7	731	174

TABLE 2: Runtime (milliseconds) and performance of the three implementations.

Graph Name	FB Alg.	Coloring Alg.	Trimming steps	FB steps	Speedup vs. Sequential	Speedup vs. Devshatwar et al. [8]
amazon-2008	Q_{VW}/L	Q/L	1	1	9.5x	0.9x
LiveJournal	Q_{VW}	Q/L	1	1	29.6x	1.0x
Flickr	Q_{VW}/L	Q/L	1	1	15.2x	1.0x
R-MAT	Q_{VW}	Q/L	1	1	47.8x	1.0x
cit-Patents	Q/L	Q/L	1	1	33.5x	1.0x
Random	Q_{VW}	Q/L	0	1	48.7x	1.1x
Pokec	Q_{VW}	L	1	1	40.7x	**1.3x**
Language	L	Q	0	2	3.4x	**1.3x**
Baidu	L	L	3	1	11.6x	**1.4x**
Pre2	L	L	0	1	6.7x	**1.6x**
CA-road	L	L	0	1	2.8x	**2.1x**
web-Berkstan	L	L	FULL	17	0.1x	**2.4x**
SSCA8	L	L	0	1	9.1x	**2.5x**
trec-w10g	L	L	2	20	0.1x	**5.6x**
Fullchip	L	L	0	1	7.6x	**7.0x**
USA-road	L	Q	0	1	3.3x	**10.5x**
Wiki-Talk	L	L	5	1	8.0x	**25.9x**

TABLE 3: Parametrization results and performance comparison.

The proposed implementation provides similar performance compared to Devshatwar et al. for the first six graphs of the dataset, while it reports speedup up to 26 times for the other graphs. This is due to the parametric feature of the proposed approach, which allows properly combining the quadratic and linear algorithms and tuning the algorithm iterations for each step i.e. trimming (I_t parameter), forward-backward (I_f parameter), and coloring. In particular, graphs with low average diameter, such as *Flickr, R-MAT, cit-Patents, Random*, show good performance also with the quadratic traversal algorithms due to less overhead compared to the linear approach that maintains frontier data queues.

The *LiveJournal* graph presents the same average diameter of *Baidu* but shows different SCC characteristics. *LiveJournal* has a very large SCC and a small percentage of trivial components, while *Baidu* the opposite. In this case, a high number of vertices with out-degree equal to zero (22.7%) favours quadratic parallelisation and one iteration of trimming.

The *amazon-2008* graph, even though it has a middle-sized average diameter, shows the best results with the quadratic approach. This is due to its very small size (*amazon-2008* is the second smallest graph in the dataset). The *Language* graph has similar size but it has a high unbalanced out-degree distribution (i.e., standard deviation 20.7 versus 3.9 of *amazon-2008*) and thus the load balancing techniques implemented in the linear BFS outperforms the quadratic parallelisation of the FB algorithm.

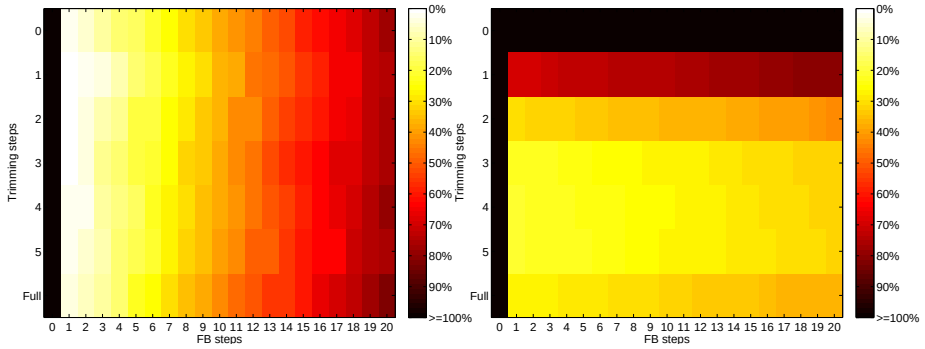
The proposed parametric implementation clearly outperforms the static Devshatwar et al. approach on graphs with high average diameter, such as *USA-Road*, and not uniform workload, such as *Wiki-Talk* (std. deviation equal to 80) thanks to the switch to the linear algorithm, which is more efficient in such a kind of graphs. Finally, both parallel implementations provide poor performance in *Web-Berkstan* and *trec-w10g* graphs due to the lack of data parallelism, which results from the small size, high diameter and low average out-degree.

We can also observe that the trimming step in road networks (*CA-Road* and *USA-Road*), *Pre2*, *Random* and *Fullchip* graphs does not significantly improve the overall performance, since the graphs contain small number of trivial SCCs. The *web-Berkstan* and *trec-w10g* require a high number of FB algorithm steps due to a high number of middle-sized SCCs. For instance, *trec-w10g* graph has the sum of the percentages of the largest SCC (29%) and trivial SCCs (31%) equal to 61% which indicates a remaining of 39% of middle-sized SCCs.

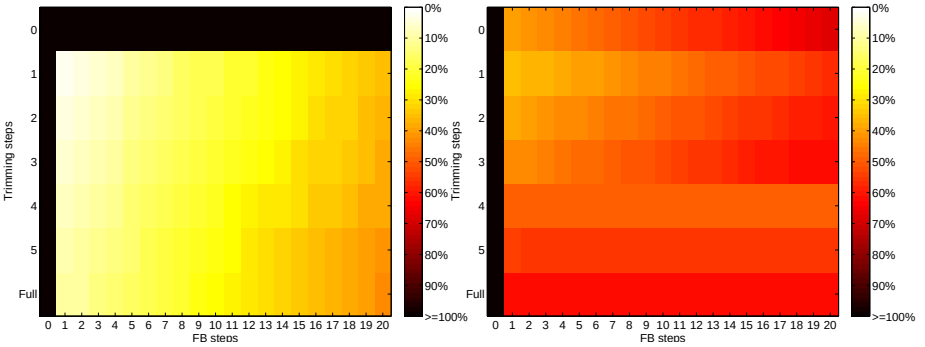
Figure 1 illustrates the impact of the parameters I_t and I_f on the overall performance for the selected graphs. The performance is represented using a color scale – lighter colors denote lower runtime. The linear parallelisation evaluated on *Amazon-2008* (Fig. 1a) shows that the performance strongly depends on the number of FB iterations (I_f set around 1 gives the best results), while the number of trimming iterations does not affect the execution time. The linear parallelisation applied to *Wiki-Talk* (Fig. 1b) shows the opposite behaviour: I_f has a very low impact on the performance, while setting a wrong I_t (e.g., I_t equal to 1 as in Deshatawar et al.) leads to 60% performance decrease. Such a different behaviour of performance over I_t and I_f relies on the different characteristics of the two graphs. *Amazon-2008* has one large SCC and a very small number of trivial SCCs, while *Wiki-Talk* has a high number of trivial SCCs. The performance of the linear parallelisation over I_t and I_f on graphs *Flickr* and *R-MAT* shows a more uniform behaviour (Fig. 1c and 1d), since the graphs have one large SCC but also a high number of trivial SCCs.

5 Conclusions

We have presented a novel parametric multi-step scheme to evaluate existing GPU-accelerated algorithms for SCC decomposition. The extensive experimental results clearly indicate that there is no algorithm that would be the best for all



(a) *Amazon-2008* (b) *Wiki-Talk*



(c) *Flickr* (d) *R-MAT*

FIG 1: Performance analysis through parametrization of I_t and I_f. Performance are represented using a color scale where lighter colors denote lower runtime.

classes of the graphs. We have dissected correlations between the internal structure of the algorithms and their performance on structurally different graphs. Our contribution, thus, represents an important step towards an ultimate solution of automatically adjusted GPU-aware algorithm for SCC decomposition.

References

[1] D. A. Bader and K. Madduri. *GTgraph: A Synthetic Graph Generator Suite*. Tech. rep. GA 30332. Georgia Institute of Technology, Atlanta, 2006.

[2] J. Barnat, P. Bauch, L. Brim, and M. Češka. "Computing Strongly Connected Components in Parallel on CUDA". In: *IPDPS'11*. IEEE Computer Society, 2011, pp. 541–552.

[3] J. Barnat and P. Moravec. "Parallel Algorithms for Finding SCCs in Implicitly Given Graphs". In: *PDMC'06*. Vol. 4346. LNCS. Springer, 2006, pp. 316–330.

[4] V. Bloemen, A. Laarman, and J. van de Pol. "Multi-core On-the-fly SCC Decomposition". In: *PPoPP'16*. ACM, 2016, 8:1–8:12.

[5] F. Busato and N. Bombieri. "BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015), pp. 1826–1838. ISSN: 1045-9219.
[6] T. A. Davis and Y. Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1.
[7] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. American Mathematical Soc., 2009.
[8] S. Devshatwar, M. Amilkanthwar, and R. Nasre. "GPU centric extensions for parallel strongly connected components computation". In: *GPGPU'16*. ACM. 2016, pp. 2–11.
[9] L. K. Fleischer, B. Hendrickson, and A. Pinar. "On Identifying Strongly Connected Components in Parallel". In: *IPDPS'00*. Vol. 1800. LNCS. Springer, 2000, pp. 505–511.
[10] P. Harish and P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *HiPC'07*. Vol. 4873. LNCS. Springer, 2007, pp. 197–208.
[11] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. "Accelerating CUDA graph algorithms at maximum warp". In: *PPoPP'11*. ACM. 2011, pp. 267–276.
[12] S. Hong, N. C. Rodia, and K. Olukotun. "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs". In: *SC'13*. ACM, 2013, 92:1–92:11.
[13] J. Kunegis. "Konect: the koblenz network collection". In: *WWW'13*. ACM. 2013, pp. 1343–1350.
[14] J. Leskovec and R. Sosič. *SNAP: A general purpose network analysis and graph mining library in C++*. `http://snap.stanford.edu/snap`. May 2016.
[15] G. Li, Z. Zhu, Z. Cong, and F. Yang. "Efficient decomposition of strongly connected components on GPUs". In: *Journal of Systems Architecture* 60.1 (2014), pp. 1 –10.
[16] X. Liu et al. "IMGPU: GPU Accelerated Influence Maximization in Large-Scale Social Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 25.1 (2014), pp. 136–145.
[17] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. "Finding Strongly Connected Components in Distributed Graphs". In: *Journal of Parallel and Distributed Computing* 65.8 (2005), pp. 901–910.
[18] D. Merrill, M. Garland, and A. Grimshaw. "Scalable GPU Graph Traversal". In: *PPoPP'12*. ACM, 2012, pp. 117–128.
[19] S. Orzan. "On Distributed Verification and Verified Distribution". PhD thesis. Free University of Amsterdam, 2004.
[20] G. M. Slota, S. Rajamanickam, and K. Madduri. "BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems". In: *IPDPS '14*. IEEE Computer Society, 2014, pp. 550–559.
[21] R. Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.