Computing Strongly Connected Components in Parallel on CUDA

Jiří Barnat, Petr Bauch, Luboš Brim, and Milan Češka
Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic

***Abstract*—The problem of decomposing a directed graph into its strongly connected components is a fundamental graph problem inherently present in many scientific and commercial applications. In this paper we show how some of the existing parallel algorithms can be reformulated in order to be accelerated by NVIDIA CUDA technology. In particular, we design a new CUDA-aware procedure for pivot selection and we adapt selected parallel algorithms for CUDA accelerated computation. We also experimentally demonstrate that with a single GTX 480 GPU card we can easily outperform the optimal serial CPU implementation by an order of magnitude in most cases, 40 times on some sufficiently big instances. This is an interesting result as unlike the serial CPU case, the asymptotic complexity of the parallel algorithms is not optimal.**

I. Introduction

Fundamental graph algorithms such as breadth first search, spanning tree construction, shortest paths, etc., are building blocks to many applications. Serial implementations of these algorithms become impractical for extremely large graphs to be processed in many application domains. As a result, parallel algorithms for processing of large graphs have been devised to efficiently use compute clusters and multi-core architectures. The transformation of a serial algorithm into a parallel algorithm is not necessarily an easy task. For example, there is most likely not an efficient parallel solution to the DFS problem [23]. Even if the algorithmic shift to parallel processing can be done, serial codes still need to be rewritten to take proper advantage of parallel processing. This is especially the case of recently introduced general purpose graphics processing units (GP GPUs). These devices contain hundreds of arithmetic units and can be harnessed to provide tremendous acceleration for many computation intensive scientific applications. The key to effective utilization of GP GPUs for scientific computing is the design and implementation of efficient data-parallel algorithms that can scale to hundreds of tightly coupled processing units. The use of several GPUs at a coarser level of parallelism can bring even more computational power.

Implementations of most of the fundamental graph algorithms on the GPU using the CUDA programming model have been reported and high performance of these implementations on very large graphs has been experimentally confirmed for example in [19] and [20].

In this paper we focus on the problem of decomposing a directed graph into its strongly connected components (*SCC decomposition*) that has not been considered so far. This problem has many applications leading to very large graphs and requiring high performance processing. One example is the web analysis based on web archives, such as topic tracking, time-frequency analysis of blog postings, and web community extraction. A particular application we also have in mind is automated verification of software (model checking, dataflow analysis, bad cycle detection, etc.), where SCC decomposition is typically used as a sub-procedure and its fast performance is crucial for assuring overall efficiency of verification tools.

Parallel SCC decomposition is a particularly tricky problem. The reason is that the (optimal) serial algorithm strongly relies on the depth-first search post ordering of vertices whose computation is known to be P-complete [23] and thus, difficult to be computed in parallel. Hence, different approaches suitable for parallel processing have been considered. See e.g. [1], [14], [18] for algorithm that works in $O(log^2 n)$ time, but requires $O(n^{2.376})$ parallel processors, or [25] for randomized parallel algorithm for the problem.

In this paper we show how selected parallel SCC decomposition algorithms (namely [17], [22], [10], [8]) can be modified in order to be accelerated on a vector processing SIMD architecture. In particular, we design a new CUDA-aware procedure for pivot selection and we adapt the parallel algorithms in order to allow for CUDA accelerated computation.

We experimentally demonstrate that with a single GTX 480 GPU we were able to gain speedup ranging from 5 to 40 when compared to the optimal sequential Tarjan's algorithm.

II. Preliminaries

A. Notation and Basic Definitions

A directed graph G is a pair (V, E), where V is a set of vertices, and $E \subseteq V \times V$ is a set of directed edges. If $(u, v) \in E$, then v is called immediate successor of u, and u is called immediate predecessor of v. The *in-degree* and *out-degree* of a vertex v is the number of immediate predecessors and successors of v, respectively. $G^T = (V, E^T)$, the *transposed graph* of $G = (V, E)$, is the graph G with all edges reversed, i.e., $E^T = \{(u, v) \mid (v, u) \in E\}$.

Let $G = (V, E)$ be a directed graph. We say that a vertex $t \in V$ *is reachable* from a vertex $s \in V$ if $(s, t) \in E^*$ where E^* denotes a reflexive and transitive closure of E. A graph is *rooted* if there is an initial vertex $s_0 \in V$ such that all

This work has been partially supported by the Czech Grant Agency grants No. 201/09/P497, 201/09/1389, 102/09/H042.

vertices in V are reachable from s_0. Given a graph G, we use n and m to denote the number of vertices and edges in G, respectively.

A set of vertices $C \subseteq V$ is *strongly connected*, if for any two vertices $u, v \in C$, we have that v is reachable from u. A *strongly connected component* (SCC) is a *maximal* strongly connected set $C \subseteq V$, i.e. such that no C' with $C \subsetneq C' \subseteq V$ is strongly connected. A maximal strongly connected component C is *trivial* if C is made of a single vertex c and $(c, c) \notin E$, and is *non-trivial* otherwise. Furthermore, C is called *leading* or *terminal* if $(V \times C) \cap E = \emptyset$ or $(C \times V) \cap E = \emptyset$, respectively. To decompose a graph into SCCs means to classify vertices of the graph according to the strongly connected component they belong to. The standard sequential algorithmic solution to the problem is due to Tarjan [24] who gave an $O(n+m)$ depth-first traversal procedure to output the list of all SCCs for a given directed graph. A subgraph of a directed graph $G = (V, E)$ given by a set of vertices $V' \subseteq V$ is a directed graph $G' = (V', E \cap (V' \times V'))$. We say that a subgraph $G' = (V', E')$ of G respects strongly connected components of G (is *SCC-closed*) if for every strongly connected component C of G we have $C \cap V' \neq \emptyset \implies C \subseteq V'$.

For $v \in W \subseteq V$, the *forward closure* of v in W is the set of reachable states from v in the subgraph of G given by W. If W is not specified, $W = V$. The forward closure of $S \subseteq W$ in W is the union of forward closures in W over all vertices from S. Finally, the *backward closure* is defined as the forward closure in the graph G^T.

B. Parallel SCC Decomposition Algorithms

In this section we describe in more details the basic ideas behind the parallel SCC decomposition algorithms. These algorithms were designed assuming parallelism provided by shared-memory multi-core or distributed-memory architectures, hence, they need to be revised to benefit from SIMD architecture parallelism the modern GPUs offer.

1) FORWARD-BACKWARD *(*FB*) algorithm:* The FB algorithm [17] introduces the basic concept that all the other presented algorithms build on. The algorithm 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 all respect strongly connected components. These subgraphs are 1) the strongly connected component with the pivot, 2) the subgraph given by vertices in the forward closure but not in the backward closure, 3) the subgraph given by vertices in the backward closure but not in the forward closure, and 4) the subgraph given by vertices that are neither in the forward nor in the backward closure. The subgraphs that do not contain the pivot form three independent instances of the same problem, and therefore, they are recursively processed in parallel with the same algorithm. The pseudo-code of the algorithm is listed as Algorithm 1.

Algorithm 1 FB Algorithm

```
proc FB($V$)
1:  if $V \neq \emptyset$ then
2:      $pivot \leftarrow PIVOT(V)$
3:      $F \leftarrow$ FWD($pivot$, $V$)
4:      $B \leftarrow$ BWD($pivot$, $V$)
5:      $F \cap B$ is SCC
6:      in parallel do
7:          FB($F \setminus B$)
8:          FB($B \setminus F$)
9:          FB($V \setminus (F \cup B)$)
10:     end in parallel
11: end if
```

Algorithm 2 COLORING Algorithm

```
proc COLORING($V$)
1:  if $V \neq \emptyset$ then
2:      $PredList, (V_k)_{k \in PredList} \leftarrow$ FWD-MAXCOLOR($V$)
3:      for all $k \in PredList$ do
4:          in parallel do
5:              $B_k \leftarrow$ BWD($k$, $V_k$)
6:              $B_k$ is SCC
7:              COLORING($V_k \setminus B_k$)
8:          end in parallel
9:      end for
10: end if
```

Practical performance of the algorithm may be further improved by performing elimination of leading and terminal trivial strongly connected components – the so called TRIMMING [21]. The TRIMMING procedure builds upon a topological sort elimination. The key idea is as follows. 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 and forward and backward closures are computed. The removal of a vertex may, however, render another vertex or vertices to have zero in-degree (out-degree). Therefore, the elimination is iteratively repeated until no more vertices with zero in-degree (out-degree) exist. Only after that, the pivot is selected and the algorithm proceeds as stated above. Note that the elimination procedure is also referred to as OWCTY (from One-Way-Catch-Them-Young algorithm) elimination procedure and has been used also for other graph related problems, see e.g. [6], [16].

2) COLORING/HEADS-OFF *algorithm:* The main limitation of the FB algorithm is that it performs $O(m+n)$ work to detect a single strongly connected component. This may be rather expensive strategy if the given graph contains many small but non-trivial components. Completely opposite approach is taken in the algorithm COLORING [22]. Algorithm COLORING is capable of detecting many strongly connected components in a single recursion step, however, for the price of $O(n.(m+n))$ procedure. The idea of the algorithm relies on the propagation of unique and totally ordered identifiers (colors) associated with vertices. Initially, each vertex keeps its own color. The colors are then iteratively propagated along edges of the graph (procedure FWD-MAXCOLOR) 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 fix-point is reached (no color update is possible), the colors associated with vertices partition the graph into multiple component respecting subgraphs. All vertices of a subgraph are reachable from the vertex whose color is associated with vertices in the subgraph, and this vertex lies in the leading strongly connected component of the subgraph. Therefore, the related component can be identified by computing a backward closure of the vertex restricted to the subgraph. This is what the algorithm does for all the subgraphs in parallel prior the recursion step. See the pseudo-code as listed in Algorithm 2. Let us also note that the propagation procedure is rather expensive if there are multiple large components in the graph [9].

3) RECURSIVE OBF algorithm: Similarly to COLORING algorithm, also the OBF procedure [10] aims at decomposing the graph in more than three component respecting subgraphs within a single recursion step. However, unlike the COLORING algorithm, the price of OBF procedure is $O(m+n)$. The name OBF is an acronym of the three procedures the algorithm comprises of: OWCTY elimination and Backward and Forward reachability.

To identify the subgraphs (*OBF slices* in terminology of RECURSIVE OBF algorithm) of a rooted chunk (subgraph reachable from a single vertex) the procedure iteratively employs the following three steps until the whole graph is processed:

- O Apply OWCTY elimination to remove leading trivial strongly connected components (TRIMMING), and return vertices that were not eliminated, but some of their immediate predecessors were.
- B Compute backward closure of vertices returned in the previous O step, vertices in the closure form a subgraph (slice) denoted by B.
- F Compute forward closures of vertices returned in the previous O step within the subgraph given by B in order to remove the slice B from the graph. The immediate successors of vertices in B that are outside B are identified as new initial states (*Seeds*) for the rest of the graph.

Should the subgraphs (slices) be processed recursively by RECURSIVE OBF [8], [9] they first need to be split into rooted chunks. For the pseudo-code of the algorithm see Algorithm 3. The recursion stops when the subgraph to be processed recursively is formed by a single strongly connected component.

Algorithm 3 RECURSIVE-OBF Algorithm

```
1: while $V \neq \emptyset$ do
2:    Pick a vertex $v \in V$
3:    $Range \leftarrow FWD(v, V)$
4:    $Seeds \leftarrow \{v\}$
5:    $V \leftarrow V \setminus Range$
6:    in parallel do
7:       OBF-X($Seeds, Range$)
8:    end in parallel
9: end while
```

Procedure 4 OBF-X($Seeds, Range$)

```
1: $Original\_Range \leftarrow |Range|$
2: while $Range \neq \emptyset$ do
3:    $Elim, Reached, Range \leftarrow$ OWCTY($Seeds, Range$)
4:    All elements of $Eliminated$ are trivial SCCs
5:    $B \leftarrow BWD(Reached, Range)$
6:    if $|B| = Original\_Range$ then
7:       $B$ is SCC
8:    else
9:       in parallel do
10:         RECURSIVE-OBF($B$)
11:      end in parallel
12:      $Seeds \leftarrow FWD\_SEEDS(B, Range)$
13:   end if
14:   $Range \leftarrow Range \setminus B$
15: end while
```

III. CUDA ARCHITECTURE

The Compute Unified Device Architectures (CUDA) [15], developed by NVIDIA, is a parallel programming model and a software environment providing general purpose programming on Graphics Processing Units. At the hardware level, GPU device is a collection of multiprocessors each consisting of eight scalar processor cores, instruction unit, on-chip shared memory, and texture and constant memory caches. Every core has a large set of local 32-bit registers but no or a very small cache (L1 cache has configurable size of 16-48KB). The multiprocessors follow the SIMD architecture, i.e., they concurrently execute the same program instruction on different data. Communication among multiprocessors is realized through the shared device memory that is accessible for every processor core.

On the software side, the CUDA programming model extends the standard C/C++ programming language with a set of parallel programming supporting primitives. A CUDA program consists of a *host* code running on the CPU and a *device* code running on the GPU. The device code is structured into the so called *kernels*. A kernel executes the same scalar sequential program in many *independent data-parallel threads*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment on all multiprocessors a group of threads called a *warp* executes instructions in a lock-step manner. When several warps are scheduled on

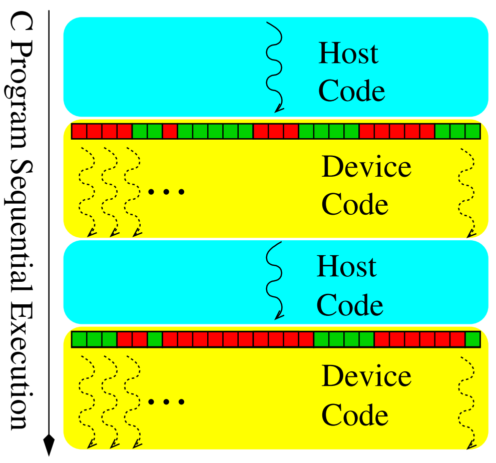

Figure 1. Sequential heterogeneous computation workflow with CUDA.

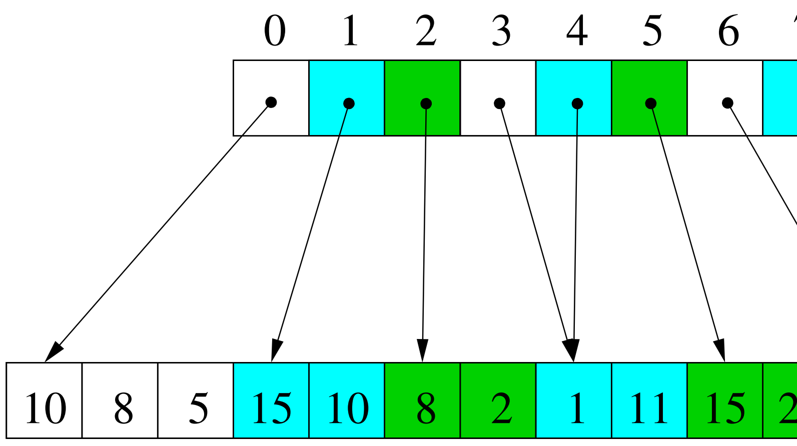

Figure 2. Adjacency list representation: a graph $G = (V, E)$ is stored as two arrays of sizes $|V| + 1$ and $|E|$.

a multiprocessor, memory latencies and pipeline stalls are hidden primarily by switching to another warp. Overall the combination of out-of-order CPU and data-parallel processing GPU allows for heterogeneous computation as illustrated in Figure 1, where sequential host code and parallel device code are executed in turns.

IV. Data-Parallel SCC Decomposition

Instead of trying to devise a completely new algorithm for SCC decomposition that would be primarily suited for the CUDA architecture, we decided for a different course of action. And that not to change the provably correct layout of the existing parallel algorithms, but instead force both the underlying graph representation and the incorporated primitive graph operations to assume and enable vector processing. Hence once the representation and graph primitives (we are adopting both concept and name from numerical vector primitives from [11] for graph related setting) are prepared, we may start building the respective algorithms with relative ease.

A. Data representation

Data structures used for CUDA accelerated computation must be designed with care. First, they have to allow independent thread-local data processing so that the CUDA hardware can employ massive parallelism. And second, they have to be small so that the high latency device-memory access and limited device-memory bandwidth are not large performance bottlenecks. As for the SCC decomposition algorithm, it is the adjacency list representation of the graph G to be encoded appropriately in the first place. Note that uncompressed matrix or dynamically linked adjacency list violate the requirements and as such they are inappropriate for CUDA computing. We encode the graph as an adjacency list that is represented as two one-dimensional arrays, similarly as in [19]. One array keeps target vertices of all the edges of the graph. The target vertices stored in the array are ordered according to the source vertices of the corresponding edges. The second array then keeps an index to the first array for every vertex in the graph. Every index points to the position of the first edge emanating from the corresponding vertex. See Figure 2. If other data associated to a vertex are needed by a CUDA kernel algorithm, then they are organized in vectors as well.

B. Computation of closures

The core procedure used in all of the algorithms is the computation of forward and backward closure (see Algorithm 7). The result of the computation of a closure procedure is a vector *visited* of $|V|$ bits indicating which of the vertices belong to the closure or not. Initially, only the bits for the vertices of which closure should be computed are set to one. To compute the closure we employ a CUDA kernel (Algorithm 5) in which we define a separate thread for every vertex. In the case of the forward closure, each thread checks if the corresponding vertex is within the closure set, and if so, it sets the presence bit for all immediate successors of the vertex. Quite often, the computation of the closure set needs to be restricted to a subgraph. To that end we denote each subgraph with a unique number and use other date structures of size $O(|V|)$ to identify the subgraph each vertex belongs to. The thread in the closure kernel then updates the presence bit of a successor only if it is a part of the same subgraph as the source vertex. In the rest of the paper we will use an equivalence relation $\sim$ to denote that two vertices are part of the same subgraph, and $[x]$ to denote vertices equivalent to a vertex x, i.e. $[x] = \{v \in V \mid x \sim v\}$.

There are two options to compute the backward closure. Either we can compute the representation of the transposed graph and employ the forward closure CUDA kernel, or we can devise a separate kernel in which each thread checks the presence bits of immediate successors of its vertex and then if some of them are in the closure set, it sets the presence bit for its own vertex. Again this can be done with respect

Algorithm 5 F-KERNEL$(G,\ \mathit{visited},\ \mathit{terminate})$

```
For all $v \in V$:
1: if $(\mathit{visited}[v] = \texttt{true})$ then
2:     for all $u \in V.\ (v,u) \in E$ do
3:         if $v \sim u\ \wedge\ \mathit{visited}[u] = \texttt{false}$ then
4:             $\mathit{visited}[u],\ \mathit{terminate} \leftarrow \texttt{true, false}$
5:         end if
6:     end for
7: end if
```

Algorithm 6 B-KERNEL'$(G, \mathit{visited},\ \mathit{terminate})$

```
For all $v \in V$:
1: if $\mathit{visited}[v] = \texttt{false}$ then
2:     if $\exists u \in V.\ (v,u) \in E$ then
3:         if $v \sim u\ \wedge\ \mathit{visited}[u] = \texttt{true}$ then
4:             $\mathit{visited}[v],\ \mathit{terminate} \leftarrow \texttt{true, false}$
5:         end if
6:     end if
7: end if
```

Algorithm 7 FWD-REACH - host code

```
In: $G = (V,E),\ P \subseteq V$
Out: $\forall v \in V : \mathit{visited}[v] = \mathit{true} \Leftrightarrow \exists u \in P :\ (u,v) \in E^*$
1: for all $u \in P$ do
2:     $\mathit{visited}[u] \leftarrow \texttt{true}$
3: end for
4: $\mathit{terminate} \leftarrow \texttt{false}$
5: while $\mathit{terminate} = \mathit{false}$ do
6:     $\mathit{terminate} \leftarrow \texttt{true}$
7:     F-KERNEL$(V,\ \mathit{visited},\ \mathit{terminate})$
8: end while
```

to a particular subgraph. While obviously the latter solution is more space efficient, our experiments have shown almost exclusive performance dominance of the first solution, hence we opted for the former. For the difference between the approaches, see pseudo-codes as listed in Algorithms 5 and 6.

A common drawback of most CUDA kernels for graph procedures is that many threads read some data from memory, but after evaluating them they do not write any data back. For example, in the case of the closure procedures each thread accesses the vector of presence bits and if it reads zero for its corresponding vertex, it terminates without making any update to the vector. As a result, the CUDA hardware has to perform a lot of useless and expensive memory read operations. A possible solution [20] to the problem is to reorganize threads so that only those threads are deployed that actually do some update to the memory. However, this preprocessing is quite an expensive procedure and does not lead to a consistent speed-up. Therefore, we have devised a different solution to the problem. We maintain an additional vector of $\lceil |V|/32 \rceil$ elements where we keep an information which warps (32 consecutive threads) will perform an update to the memory in the succeeding iteration. Namely, if all vertices processed within a single warp are not part of

Algorithm 8 TRIM-KERNEL$(G,\ \mathit{eliminated},\ \mathit{terminate})$

```
For all $v \in V$:
1: if $\mathit{eliminated}[v] = \texttt{false}$ then
2:     $\mathit{elim} \leftarrow \texttt{true}$
3:     if $\exists u \in V.\ (u,v) \in E\ \wedge\ v \sim u$ then
4:         $\mathit{elim} \leftarrow \texttt{false}$
5:     end if
6:     if $\mathit{elim} = \texttt{true}$ then
7:         $\mathit{eliminated}[v],\ \mathit{terminate} \leftarrow \texttt{true, false}$
8:     end if
9: end if
```

the closure set (all have the presence bit set to zero) no update to memory will occur due to this warp. By employing special *broadcast* operation available in CUDA we can thus replace (potentially up to) two 128-byte and one 64-byte data transactions with a single 32-byte memory read operation followed by the broadcast to all threads in the warp. According to our experiments this approach led to an observable speedup in many cases while introducing minimal slowdown in the other ones.

C. TRIMMING *and self-loop detection*

As explained in Section II, some algorithms employ TRIMMING procedure to efficiently deal with leading and terminal trivial SCCs. The goal of the procedure is to identify vertices of the underlying subgraph that have no immediate predecessors (in the case of leading components) or immediate successors (in the case of terminal components) in the subgraph. Such vertices may be iteratively removed from the subgraph as trivial SCCs. The host code of the TRIMMING procedure is quite similar to the host code of forward closure (listed as Algorithm 7) and therefore we list only the TRIMMING kernel, see Algorithm 8. The result of the procedure is a vector *eliminated* of $|V|$ bits indicating which of the vertices have been eliminated. Note that the procedure can be easily augmented also to eliminate SCCs made of a single vertex with a self-loop by simply ignoring the self-loop edges.

D. Pivot selection

There are several stages of the algorithms that require a single vertex to be chosen within a processed subgraph – the so called pivot. Pivot selection plays significant role in practical performance of the algorithms. As a good heuristics to pivot selection the algorithms typically rely on a pseudo-random number generator. In our approach, we not only need to select a single pivot, but since we share kernels for graph procedures over multiple subgraphs, we need to choose a number of pivots, one for each subgraph. To that end, the usage of a random number generator seems inappropriate as we cannot guarantee that after a repeated random selection, the selected vertices will satisfy the desired distribution.

We have, therefore, opted for a different solution. The basic idea of our pivot selection is to let all vertices of a subgraph concurrently write their own unique identifiers to a single memory location. After that the location keeps a single value that identifies the pivot. Surprisingly, the most challenging problem when implementing the idea was where to define/store the memory location for a subgraph. Note that within a single kernel we may select pivots for quite a large number of subgraphs.

To solve the problem we employed the observation that a subgraph when defined is fully contained within a *parent* subgraph (the part of the graph from whose subgraphs the pivots are voted). For our first solution to the problem suppose now that the pivot of the parent subgraph has an extra space allocated to it. Then all the child subgraphs may be learnt about the pivot of their parent subgraph and thus they may use the extra space allocated to the parent subgraph pivot as the memory location they need for their own pivot selection. If there are multiple child subgraphs of one parent subgraph then they are serialized for the usage of the memory location. Since we do not know in advance which vertices become pivots, we reserve extra space for every vertex. This requires at least $|V|size(v)$ additional space, where $size(v)$ is the space necessary for identification of a single vertex.

In our second approach to the problem we have allocated a single shared vector of memory locations and make sure that every computed subgraph gets a unique pointer to the vector. If each recursive step defines bounded number of subgraphs, we can compute the unique number of a child subgraph from the unique number associated with the parent subgraph. For example, in the case of FB algorithm, the bound is equal to three, so the three new subgraphs of a parent subgraph with unique number i will get numbers $3i + 0$, $3i + 1$, and $3i + 2$. An obvious problem of the second solution is that the number of subgraphs is unknown in advance, hence the unique numbers associated with the subgraphs may grow beyond the size of the preallocated vector. Note that if that happens a lot of unique numbers of subgraphs that were parent subgraphs before, are unused. We therefore postpone the computation of the algorithm and run a heuristics that renumbers active subgraphs so that they get numbers somewhere at the beginning of the vector. To compute new unique numbers of active subgraphs we employ hash function. Collisions due to the hash function are relatively rare, and they are handled sequentially after the renumbering by the hash function.

E. Processing of independent subgraphs

Within the scope of SCC decomposition the computation of forward or backward closures are typically restricted to a particular component respecting subgraph of the original graph. As soon as the algorithm is deeper in its recursion, the same procedures are typically executed over different subgraphs. If each operation such as the computation of a forward closure, is implemented as a CUDA kernel, we can easily mimic the recursion as suggested by the algorithms within the host code (we let the host code call a separate kernel for each graph operation over every subgraph in every recursion branch). However, if a kernel is executed in this approach over a whole matrix, a lot of CUDA threads, namely those that are deployed for vertices out of the processed subgraph, are idling or performing useless work. We can avoid this inefficiency if we deploy only the threads for vertices of the subgraph, but to be able to do so we would have to renumber the vertices of the graph so that the vertices of the subgraph are well-distributed in the vector of vertices, i.e. at least in a number of continuous blocks. This renumbering would of course kill any benefit the preprocessing might have brought.

Algorithm 9 FB Algorithm - host code

```
In: Directed graph $G = (V, E)$
Out: SCC decomposition of $G$
$u \sim v \Leftrightarrow range[u] = range[v]$
1: while $terminate$ = false do
2:     FWD-REACH($G$, $pivots$, $visited.f$)
3:     BWD-REACH($G$, $pivots$, $visited.b$)
4:     TRIMMING($G$, $elim$)
5:     PIVOT-SEL($pivots$, $range$, $visited$, $elim$)
6:     UPDATE($range$, $visited$, $elim$, $terminate$)
7: end while
```

We therefore proceed in a different way and share the calls to the kernels that are made for the same operation over different subgraphs in different recursion branches. In particular, if we synchronize the recursion of the algorithm so that in the second recursion step, let us say, the computation of a forward closure is executed simultaneously over multiple independent subgraphs, we can employ a single CUDA kernel to compute all the forward closures at the same time. This synchronization over recursion deepening and kernel sharing principles allow us to reformulate the recursion present in the algorithms by means of iterative procedures (while loops). This is exemplified on pseudocode for FB algorithm listed in Algorithm 9 (though the idea is common to all implemented algorithms). According to our experience the penalty for explicit synchronization due to loop iterations is easily outweighed by performance gain achieved due to the kernel sharing.

V. Accelerating Algorithms on CUDA

We have prepared all the representation details and graph primitives to be applied on graphs divided potentially into multiple subgraphs. Thus the idea foreshadowed above of mapping the recursive nature of the presented algorithms into iterative processing, enables us to implement the algorithms on various vector models of computation, e.g. on the heterogeneous model (presented in Section III) that all CUDA-equipped off-the-shelf computers possess.

Algorithm 10 COL-KERNEL$(G,\ map,\ pivots,\ inner)$

```
For all $v \in V$:
1: $map[v] \leftarrow \max\{v,\ map[v]\}$
2: for all $u \in V.\ (u,v) \in E$ do
3:     if $v \sim u\ \wedge\ map[v] < map[u]$ then
4:         $map[v],\ inner \leftarrow map[u],\ \texttt{true}$
5:     end if
6: end for
7: if $map[v] = v$ then
8:     $pivots[v] \leftarrow \texttt{true}$
9: end if
```

A. FB *algorithm on CUDA*

Once the algorithm is given as iterative procedure, the adaptation for CUDA environment is rather straightforward. See the pseudo-code as listed in Algorithm 9. We are using the vector *visited* to indicate which of the vertices belong to the forward respectively backward closure (*visited.f*, *visited.b*), vector *elim* to keep the eliminated vertices and vector *pivots* to determine the pivots for next iteration of the algorithm. Finally, the vector *range* is used to identify the subgraph that each vertex belongs to. The UPDATE kernel recomputes the *range* vector (the relation $\sim$) according to the vectors *visited* and *elim*. Moreover, it sets the variable *terminate* to `true` if all vertices from previous iteration were either visited by both the forward and backward closure procedure, or were eliminated.

B. COLORING *algorithm on CUDA*

Likewise the FB algorithm, also the COLORING algorithm can be formulated as an iterative algorithm. In such a case every loop iteration consists of two procedures: the color-propagation procedure that partitions the graph into multiple subgraphs, and backward closure procedure that identifies and removes the leading component of every subgraph. We list pseudo-code of the CUDA kernel for the color propagation only, see Algorithm 10, as the host code has the similar structure to the host code of algorithm FB listed as Algorithm 9. Note that the color propagation procedure also computes the vertex (pivot) for the succeeding backward closure procedure, and that variable *inner* is used to detect that no fix-point has been reached yet.

C. OBF *algorithm on CUDA*

Unlike the case of FB and COLORING algorithms, the adaptation of the OBF algorithm to the CUDA environment was a little bit more involved. In our final solution, we have decided not to use three independent CUDA kernels for individual phases (O, B, and F), but instead we have devised a single CUDA kernel that performs all three phases at the same time. Every vertex keeps extra information to know in which phase it is currently processed. See Algorithms 11, and 12.

Algorithm 11 CUDA OBF Algorithm - host code

```
In: Directed graph $G = (V, E)$
Out: SCC decomposition of $G$
$u \sim v \Leftrightarrow range[u] = range[v]$
1: while $terminate = false$ do
2:     while INTERRUPTION$(i) = false$ do
3:         OBF-KERNEL$(O, B, F, V)$
4:     end while
5:     if $phase[i] \in O$ then
6:         UPDATE_O$(i)$
7:     end if
8:     if $phase[i] \in F$ then
9:         UPDATE_F$(i)$
10:    end if
11:    if $phase[i] \in B$ then
12:        UPDATE_B$(i)$
13:    end if
14:    UPDATE$(range,\ elim,\ terminate)$
15: end while
```

Algorithm 12 OBF-KERNEL$(G,\ phase,\ reach,\ elim)$

```
For all $v \in V$:
1: if $phase[v] = O\ \wedge\ reach.o[v] = \texttt{true}$ then
2:     if $(\forall u \in V.(u,v) \in E\ \wedge\ u \sim v)$ then
3:         for all $w \in V.\ (v,w) \in E\ \wedge\ v \sim w$ do
4:             $reach.o[w] = \texttt{true}$
5:         end for
6:         $elim[v] = \texttt{true}$
7:     end if
8: else
9:     if $phase[v] = B$ then
10:        for all $u \in V.\ (u,v) \in E\ \wedge\ u \sim v$ do
11:            $reach.b[u] = \texttt{true}$
12:        end for
13:    else
14:        for all $u \in V.\ (v,u) \in E\ \wedge\ u \sim v$ do
15:            $reach.f[u] = \texttt{true}$
16:        end for
17:    end if
18: end if
```

The OBF-KERNEL proceeds until one of the phases terminates, which is detected by the procedure INTERRUPTION. After the termination, vertex i is returned to identify the subgraph of the phase that has terminated and caused the interruption. An update procedure is then executed according to the type of the phase.

- UPDATE_O updates not eliminated vertices of $[i]$ to be processed by the next phase (B).
- UPDATE_B checks whether the reached part of $[i]$ is rooted. If so, *range* of vertices in the reached part is set to a common unique value and the part is eliminated as a SCC. If the reached part was not rooted, we select a pivot and execute a forward closure to get a rooted subgraph. The phase of the rest of vertices in $[i]$ is set back to O. We also set $reach.o[v]$ for every vertex v

that is a successor of a reached vertex in $[i]$.

- UPDATE_F selects a pivot from the not reached part of $[i]$ to start a new forward reachability there. Simultaneously the phase is set to O for the reached vertices and the pivot of $[i]$ is the only one set to reached. Finally, the two parts (reached and not reached) are separated (within $\sim$) by setting the range of the reached vertices to a new unique value.

Procedure UPDATE merely checks whether all the vertices were eliminated (either by OWCTY or when found to be in a rooted subgraph by the UPDATE_B) and sets the $terminate$ variable accordingly.

It is clear to see that the OBF-KERNEL forces the individual threads to perform different tasks if their vertices fall into different sets. Which is in the opposition to one of the principles of CUDA programming since all threads within any *warp* should at one time perform the same instruction. This is of a little problem when the sets O, B, F (containing vertices in the respective phases) are large and consist of consecutive vertices (meaning they are stored in an uninterrupted row in the adjacency matrix representation), but as they grow smaller or less compact it might entail considerable slowdown.

We have tried to at least partially eliminate this problem by opting out subgraphs that are too small. Once the size of a subgraph drops below a given threshold, we employed COLORING algorithm to finish the decomposition of the subgraph. In order to do so, we had to adapt the OBF algorithm to compute the sizes of the produced subgraphs. Once all the subgraphs are small enough we actually stop the OBF algorithm and continue with the COLORING algorithm. Despite the inevitable overhead of the size computation, this strategy often lead to significant improvement according to our experimental measurements. Furthermore, the OBF can be augmented with an initial call to a TRIMMING procedure in order to avoid the costly OBF-like computation on all the leading and terminal trivial components.

D. Employing Multiple CUDA Devices

CUDA technology provides tremendous computing power due to the massive parallelism, however, with limited memory resources. As such its applicability to SCC decomposition is limited by the size of the graph that can fit the memory of a single CUDA device.

We have shown recently [4] how to efficiently employ multiple CUDA devices to solve the accepting cycle detection problem. More precisely, we were able to distribute computation of various elementary graph algorithms among multiple GPUs and alter the underlying graph representation to successfully overcome the space limitation. And even though our parallel algorithm was quite inter-CUDA communication intensive, we were able to preserve a decent time efficiency of the whole parallel system.

It is worthy to observe that accepting cycle detection and SCC decomposition algorithms both use the same elementary graph procedures like closure computation or unique and totally order identifiers propagation. Thus is seems tractable, due to the idea of building SCC decomposition algorithms from graph primitives, to follow the same process of devising multi GPU implementation as presented in [4].

VI. EXPERIMENTAL EVALUATION

We compare the performance of the described CUDA algorithms with the CPU implementation of the TARJAN'S algorithm that is considered to be the best sequential algorithm for SCC decomposition. For this purpose we have implemented our own highly optimized version of TARJAN'S algorithm using identical representation of adjacency list as the one used for CUDA computation. Our implementation of the TARJAN'S algorithm outperforms (2 times) the Boost [12] implementation.

We have also implemented multi-core versions of the algorithms for the standard parallel shared-memory platforms. To that end we have experimented with two implementations. In the first variant, we basically let CPU cores perform the CUDA version of each algorithm without employing CUDA device. In the second version, we took the approach of parallel distributed-memory graph traversal procedures, see e.g. [9], and we applied it to shared-memory environment. For the shared-memory message passing we used lock-free FIFO data structures, as suggested in [5]. Unfortunately, none of our implementations were able to outperform TARJAN'S algorithm using quad-core architecture, which can be explained by extremely cache-efficient representation of the graph used for TARJAN'S algorithm that was used on relatively small graphs (we have experimented with graphs whose representation fitted 1.5 GB of RAM of our CUDA GPU card). All the experiments were run on a Linux workstation with an AMD Phenom(tm) II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066 MHz RAM and NVIDIA GeForce GTX 480 GPU with 1.5 GB of GPU memory.

To evaluate the algorithms we used input graphs as generated by Georgia Tech. graph generator (GTgraph) [3] containing: Scalable Synthetic Compact Applications (SSCA) benchmark suite [2], Recursive Matrix (R-MAT) generator [13], Erdös-Rényi random graph generator; and graphs as produced by the enumerative model checker DiVinE [7].

We provide comparison of performance of the following algorithms: serial CPU-based forward reachability denoted by CPU REACH, CUDA-based forward reachability denoted by CUDA REACH, TARJAN'S algorithm, CUDA-based FB algorithm (+ TRIMMING), CUDA-based COLORING algorithm, and CUDA-based OBF algorithm (+ TRIMMING, + COLORING, + TRIMMING and COLORING). Table I lists run-times of the algorithms if executed on the three types of synthetic graphs with average degree

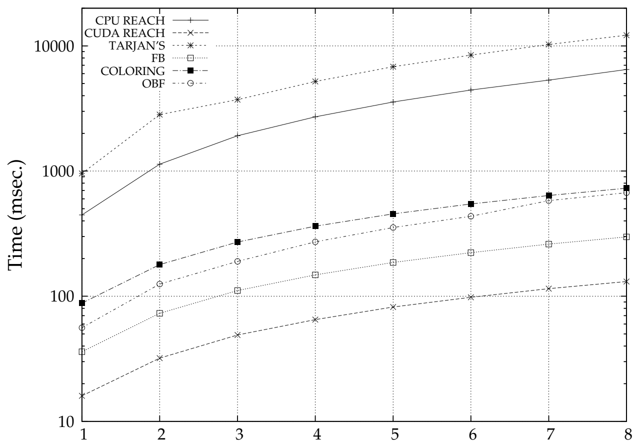

Figure 3. Run-times for Random graphs in milliseconds.

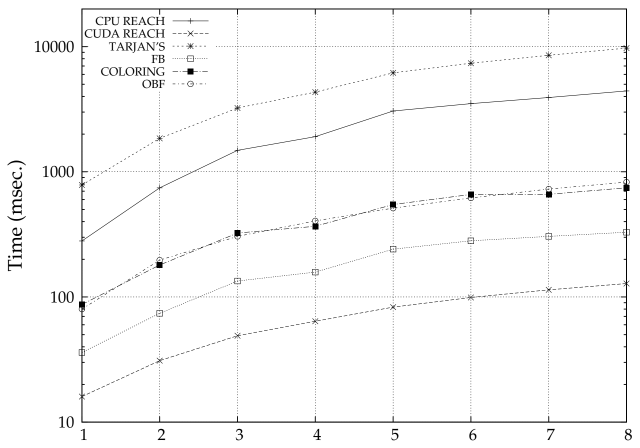

Figure 4. Run-times for R-MAT graphs in milliseconds.

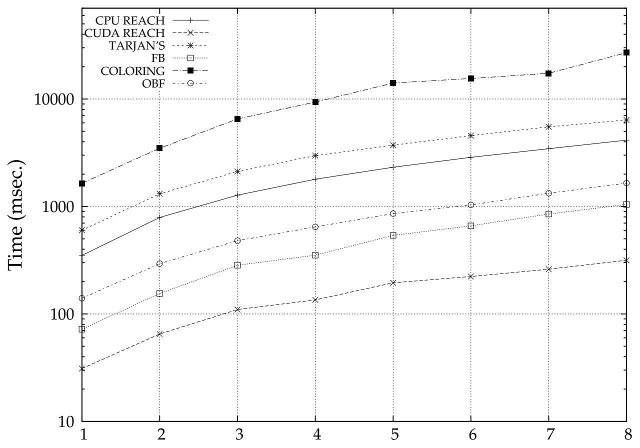

Figure 5. Run-times for SCCA#2 graphs in milliseconds.

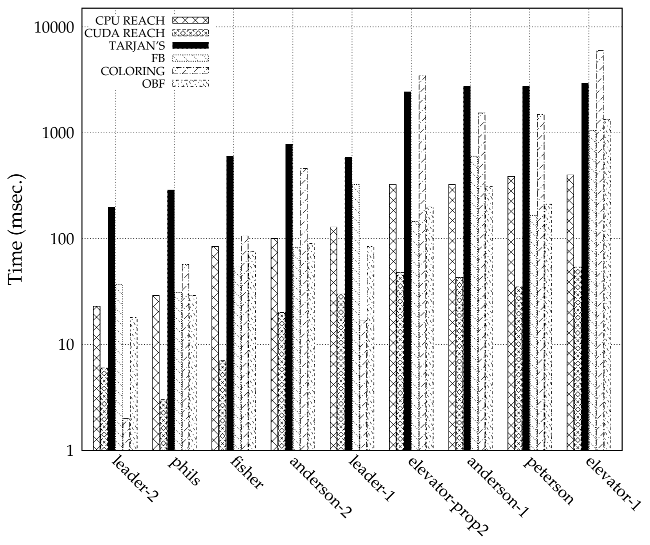

Figure 6. Run-times for model checking graphs in milliseconds.

set to twelve and scaled up by the number of vertices. Table II gives run-times of particular algorithms for graphs corresponding to model checking problems. The run-times are also plotted in Figures 3, 4, 5, and 6 using the best time available among versions of individual parallel algorithms.

Note that the computation of the algorithms which has reached 50 seconds time limit has been aborted and is marked in the tables by '-'. Also note that the reported run-times for CUDA-based forward reachability exhibit similar values to the values reported in [19] and [20].

Table III enumerates the difference in run-times of the algorithms when executed on the previous and current generation of NVIDIA graphics cards, i.e. GTX 280 and GTX 480. Again, the dashes mean that the algorithm did not finish in time (50s) on some instances. Examining the hardware specifications of the respective cards we see that the frequency of individual computation cores was increased rather modestly (from 602 to 700 MHz) when compared to doubling of the number of cores. Bearing this in mind, we may consider the observed speedup (closing to threefold in some instances) to be a witness of effective scalability of the presented data-parallel algorithms. The seeming superlinear speedup is to be accredited to both the higher frequency and unprecedented cached memory hierarchy.

Finally, Table IV gives the overall achieved speedup when the best time available among versions of individual parallel algorithms is considered.

We have observed that the performance of CUDA-based algorithms deeply depends on the average degree of the vertices in the graph. Simple reachability procedure (forward closure) performs in linear time with respect to the radius of the graph, which tends to expand as the average degree decreases. For graphs with low degree, the performance of reachability procedure may be improved using our heuristics to reduce the number of memory loads, see Subsection IV-B. Generally, we observe that the scalability and efficiency of the parallel reachability procedure effectively limits scalability and efficiency of the SCC decomposition algorithms.

Graph type	Algorithm	Number of vertices in milions, average degree 12 (Number of SCC components)								Total
		1M (16)	2M (31)	3M (30)	4M (48)	5M (61)	6M (72)	7M (97)	8M (106)	
Random	CPU REACH	446	1135	1915	2712	3563	4440	5331	6479	26021
	CUDA REACH	16	32	49	65	82	98	115	131	588
	Tarjan's	957	2825	3722	5195	6822	8443	10265	12169	50398
	FB	39	85	127	183	243	309	384	456	1826
	FB + Trim	36	73	111	148	186	223	261	298	1336
	Coloring	88	179	271	363	455	546	638	729	3269
	OBF	56	125	190	272	354	435	580	690	2702
	OBF + Col	62	129	196	284	372	459	620	741	2863
	OBF + Trim	81	165	249	334	418	502	586	671	3006
	OBF + Col + Trim	81	166	251	336	421	506	590	676	3027
		1M (0.48M)	2M (0.97M)	3M (0.97M)	4M (1.9M)	5M (1.0M)	6M (2.0M)	7M (2.9M)	8M (3.9M)	
R-MAT	CPU REACH	280	744	1484	1910	3060	3504	3921	4428	14903
	CUDA REACH	16	31	49	64	83	99	114	128	584
	Tarjan's	785	1851	3230	4332	6171	7365	8529	9738	42001
	FB	-	-	-	-	-	-	-	-	-
	FB + Trim	36	74	134	158	241	281	305	329	1558
	Coloring	87	180	324	367	548	659	660	745	3570
	OBF	-	-	-	-	-	-	-	-	-
	OBF + Col	-	-	-	-	-	-	-	-	-
	OBF + Trim	83	203	343	427	595	702	796	887	4036
	OBF + Col + Trim	80	197	305	405	513	618	728	827	3673
		1M (576)	2M (1.1K)	3M (1.7K)	4M (2.2K)	5M (2.8K)	6M (3.4K)	7M (4.0K)	8M (4.4K)	
SSCA#2	CPU REACH	350	790	1274	1794	2319	2866	3451	4141	16985
	CUDA REACH	31	65	110	135	195	223	261	316	1336
	Tarjan's	601	1313	2116	2973	3721	4565	5513	6377	27179
	FB	299	833	2089	3003	5168	7702	8455	11483	39032
	FB + Trim	72	155	284	352	538	661	851	1046	3959
	Coloring	1646	3483	6532	9373	14095	15511	17352	27020	95012
	OBF	281	913	2008	3092	4872	6939	9401	11528	39034
	OBF + Col	316	1025	2269	3536	5574	7964	10855	13223	44762
	OBF + Trim	143	310	532	724	989	1257	1539	1930	7424
	OBF + Col + Trim	140	294	481	646	859	1035	1328	1650	6433

Table I
Run-times for synthetic graphs in milliseconds.

Model (n, m, Number of SCC components)	Algorithm									
	CPU REACH	CUDA REACH	Tarjan's	FB	FB + Trim	Coloring	OBF	OBF + Col	OBF + Trim	OBF + Col + Trim
leader-2 (0.7M, 3.8M, 0.7M)	23	6	197	383	37	2	18	19	28	28
phils (0.7M, 6.0M, 59K)	29	3	287	20344	31	57	54	34	52	29
fisher (2.5M, 13.8M, 81K)	84	7	597	18206	54	106	175	76	185	84
anderson-2 (3.1M, 13.4M, 1.6M)	100	20	774	-	83	459	115	132	90	93
leader-1 (3.6M, 26.6M, 3.6M)	129	30	582	7504	324	17	363	84	379	403
elevator-2 (6.4M, 83.3M, 1)	323	48	2437	145	147	3441	199	200	251	249
anderson-1 (8.9M, 47.7M, 4.3M)	325	43	2738	-	600	1537	415	420	312	389
peterson (9.5M, 42.0M, 18K)	387	35	2740	13466	166	1487	211	224	266	279
elevator-1 (8.6M, 89.4M, 2.0M)	400	54	2933	-	1049	5969	1336	1370	1384	1375
Total	1800	246	13285	-	2491	13075	2886	2559	2947	2929

Table II
Run-times for model checking graphs in milliseconds.

Graph type	GPU device	Algorithm							
		CUDA REACH	FB	FB + TRIM	COLORING	OBF	OBF + COL	OBF + TRIM	OBF + COL + TRIM
Random	GTX 280	1501	3980	2888	5205	5447	6274	5754	5763
	GTX 480	588	1826	1336	3269	2702	2863	3006	3027
R-MAT	GTX 280	1651	-	3578	7496	-	-	7120	7190
	GTX 480	584	-	1558	3570	-	-	4036	3673
SSCA#2	GTX 280	3923	78892	10245	135003	68040	102546	16625	15186
	GTX 480	1336	39032	3959	95012	39034	44762	7424	6433
Model Checking	GTX 280	526	-	5188	23152	4604	4151	4344	4555
	GTX 480	246	-	2491	13075	2886	2559	2947	2929

Table III
RUN-TIMES SUMMED OVER RESPECTIVE GRAPH TYPES FOR THE PREVIOUS AND CURRENT GENERATION OF GPUS.

Graph type	GPU device	Speedup against CPU REACH	Speedup against TARJAN'S		
		CUDA REACH	FB	COLORING	OBF
Random	GTX 280	17.3	17.5	9.7	9.2
	GTX 480	44.3	37.7	15.4	18.7
R-MAT	GTX 280	9.0	11.7	5.6	5.9
	GTX 480	25.5	27.0	11.8	11.4
SSCA#2	GTX 280	4.3	2.7	0.2	1.8
	GTX 480	12.7	6.9	0.3	4.2
Model Checking	GTX 280	3.4	2.6	0.6	3.8
	GTX 480	7.3	5.3	1.0	5.6

Table IV
OVERALL ACHIEVED SPEEDUP FOR THE PREVIOUS AND CURRENT GENERATION OF GPUS.

We can conclusively state that in most experiments our algorithms were able to reach this limit.

Other observations are as follows. For Random graphs, where most of the vertices have similar degree, all algorithms significantly outperform (40 times) the TARJAN'S algorithm as they can effectively exploit the parallelism. R-MAT graphs have uneven degree distribution with most vertices of rather a small degree. These graphs expand slowly in each iteration and exhibit uneven load balancing and thus the performance of algorithms based on the computation of forward reachability is very poor. However, adding the TRIMMING phase drastically improves their performance and leads to overall twenty-sevenfold speedup. SSCA graphs exhibit similar degree distribution to the R-MAT graphs, but they typically contain large number of small non-trivial components, which limits the efficiency of the TRIMMING procedure (overall sevenfold speedup). For model checking graphs, the average degree of a vertex is rather small compared to the synthetic graphs, therefore the run-times are not as good as in the case of synthetic graphs (overall sixfold speedup).

For synthetic graphs, FB algorithm with TRIMMING has the best times. This is because the graphs usually contain small number of large components and large number of trivial or very small components. Such a structure of a graph causes the whole decomposition process to boil down to a few invocations of the forward and backward reachability interleaved with the TRIMMING procedure. On the other hand model-checking graphs contain in general bigger number of large components. For graphs with such a structure the OBF algorithm has the potential to significantly outperform the other ones. Finally, we observe that the COLORING algorithm exhibits rather unstable performance. While thriving on highly disconnected graphs or graphs with many small components, its performance degrades as the size of the components in the graph grows.

VII. CONCLUSIONS

We have demonstrated successful redesign of several parallel algorithms for SCC decomposition. The redesigned versions allow for computation acceleration on massively parallel hardware platforms such as CUDA. In particular, we have designed a new CUDA-aware procedure for pivot selection and reformulated the parallel SCC decomposition algorithms in order to outperform the optimal but inherently sequential TARJAN'S algorithm.

Thus while not proposing a strictly speaking new parallel algorithm for SCC decomposition we instead suggest methods allowing to map the recursive nature of the presented algorithms into iterative procedures. Hence instead of devising an ad-hoc solution for SCC decomposition on CUDA we have tried to establish more general approach to irregular graph algorithms. This approach aims at enabling implementation of algorithms on various vector models of computation and propounds how could other graph algorithms be altered to benefit from SIMD architecture.

We have also done an extensive experimental evaluation of the known algorithms on several types of graphs proving that with single GTX 480 GPU card we can easily outperform the optimal serial algorithm. The results imply that the FB algorithm is to be declared a winner among the particular algorithms having speedup up to fortyfold and rarely going below sixfold. Though COLORING reaches surprisingly good results on some instances, there are only a few of them. And finally despite achieving rather steady performance, the OBF algorithm seems to fall behind the other algorithms. This is because the random graph generators as used in our study fail to provide graphs with significant amount of nontrivial and large components. Whether this is the case of all application domains is, however, questionable.

REFERENCES

[1] N. Amato. Improved Processor Bounds for Parallel Algorithms for Weighted Directed Graphs. *Information Processing Letters*, 45(3):147–152, 1993.

[2] D. A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *HiPC*, volume 3769 of *LNCS*, pages 465–476. Springer, 2005.

[3] D. A. Bader and K. Madduri. GTgraph: A Synthetic Graph Generator Suite. Technical Report GA 30332, Georgia Institute of Technology, Atlanta, 2006.

[4] J. Barnat, P. Bauch, L. Brim, and M. Češka. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In *16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 259–266. IEEE Computer Society, 2010.

[5] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *SPIN '07: Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[6] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *FMCO*, volume 4111 of *LNCS*, pages 259–279. Springer, 2005.

[7] J. Barnat, L. Brim, L. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *CAV '06: Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer, 2006.

[8] J. Barnat, J. Chaloupka, and J. C. van de Pol. Improved Distributed Algorithms for SCC Decomposition. In *PDMC*, pages 65–80. CTIT, University of Twente, 2007.

[9] J. Barnat, J. Chaloupka, and J. C. van de Pol. Distributed Algorithms for SCC Decomposition. *To appear in Journal of Logic and Computation*, 2010.

[10] J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

[11] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[12] Boost: C++ libraries. http://www.boost.org/, September 2010.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, pages 442–446. SIAM, 2004.

[14] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix Sums and List Ranking. *Information and Computation*, 81(3):334–352, 1989.

[15] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 3.1,. http://developer.nvidia.com/object/cuda_3_1_downloads.html, September 2010.

[16] L. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In *TACAS*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[17] L. K. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. In *Parallel and Distributed Processing*, volume 1800 of *LNCS*, pages 505–511. Springer, 2000.

[18] H. Gazit and G. L. Miller. An Improved Parallel Algorithm That Computes the BFS Numbering of a Directed Graph. *Information Processing Letters*, 28(2):61–65, 1988.

[19] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.

[20] P. Harish, V. Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA, 2009.

[21] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[22] S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.

[23] J. H. Reif. Depth-First Search is Inherrently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[24] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[25] S. Warren. Finding Strongly Connected Components in Parallel Using O(log2n) Reachability Queries. In *SPAA*, pages 146–151. ACM, 2008.