FI MU

Faculty of Informatics
Masaryk University Brno

Computing Strongly Connected Components in Parallel on CUDA (full version)

by

Jiří Barnat, Petr Bauch, Luboš Brim, and Milan Češka

FI MU Report Series **FIMU-RS-2010-10**

Copyright © 2010, FI MU **July 2010**

Copyright © 2010, Faculty of Informatics, Masaryk University.
All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

Computing Strongly Connected Components in Parallel on CUDA (full version)*

Jiří Barnat, Petr Bauch, Luboš Brim, and Milan Češka
Faculty of Informatics, Masaryk University,
Botanicka 68a, 60200 Brno, Czech Republic
{xbarnat, xbauch, brim, xceska }@fi.muni.cz

July 20, 2010

Abstract

The problem of decomposition of 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 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 redesign the parallel algorithms in order to allow for CUDA accelerated computation. We also experimentally demonstrate that with a single GTX 280 GPU card we can easily outperform optimal serial CPU implementation, which is particularly interesting result as unlike the serial CPU case, the asymptotic complexity of the parallel algorithms is not optimal.

1 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 became impractical as graphs to be processed are extremely large in many application domains. As a result parallel algorithms for processing of large

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

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, most likely there is not an efficient parallel solution to the DFS problem [22]. 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 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 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 [18] and [19].

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 and requiring high performance processing. One example is 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 subprocedure and its fast performance is crucial.

Parallel SCC decomposition is a particularly tricky problem. The reason is that the (optimal) serial algorithm strongly relies on depth-first search post ordering of vertices whose computation is known to be P-complete and thus, difficult to be computed in parallel. Hence, different approaches suitable for parallel processing have been considered. See e.g. [17, 12, 1] for algorithm that works in $O(\log^2 n)$ time, but requires $O(n^{2.376})$ parallel processors, or [24] for randomized parallel algorithm for the problem.

In this paper we show how selected parallel SCC decomposition algorithms (namely [16], [21], [9], [7]) 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 redesign the parallel algorithms in order to allow for CUDA accelerated computa-

tion. We also experimentally demonstrate that with a single GTX 280 GPU card we can easily outperform optimal serial CPU implementation, which is particularly interesting result as unlike the serial CPU case, the asymptotic complexity of the considered parallel algorithms is not optimal.

2 Preliminaries

2.1 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* (*out-degree*) of a vertex v is the number of immediate predecessors (successors) of v. $G^T = (V, E^T)$, 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 transitive closure of E. A graph is *rooted* if there is an initial vertex $s_0 \in V$ such that all 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. 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 [23] 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* of v (or S) in W is the forward closure of v (or S) in W in the graph G^T.

Algorithm 1 FB Algorithm

proc FB(V)

1: **if** $V \neq \emptyset$ **then**
2: $pivot \leftarrow PIVOT(V)$
3: $F \leftarrow \text{FWD}(pivot, V)$
4: $B \leftarrow \text{BWD}(pivot, V)$
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**

2.2 Parallel SCC Decomposition Algorithms

In this section we describe in more detail the basic ideas behind the parallel SCC decomposition algorithms we have considered for acceleration on CUDA.

2.2.1 Forward-Backward Algorithm

The FB algorithm [16] 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 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.

Practical performance of the algorithm may be further improved by performing elimination of leading and terminal trivial strongly connected components – the so called *trimming* [20]. 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

Algorithm 2 COLORING Algorithm

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

forward and backward closures are computed. The removal of the vertex may however produce another vertex or vertices with 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 elimination procedure and has been used also for other graph related problems, see e.g. [15, 5].

2.2.2 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 [21]. 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 a 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 (the procedure FWD-MAXCOLOR) so that each vertex keeps only the maximum color among the initial color and the colors that have been propagated into it (maximal preceding color). After a fix-point is reached (no color update is possible), the colors associated to vertices partition the graph into multiple component respecting subgraphs. All vertices in a subgraph are reachable from the vertex of which color is the subgraph. Moreover, this vertex lies in the leading strongly connected component of the subgraph and as such the related component can be identified by performing the backward closure of the vertex restricted to the subgraph. This is what the algorithm

does for all 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 [8].

2.2.3 Recursive OBF algorithm

Similarly to COLORING algorithm, also the OBF procedure [9] 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 $O(m + n)$.

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 closure of vertices returned in the previous O step within the subgraph given by B plus vertices being immediate successors of vertices in B to remove the slice B from the graph and to identify new initial states (*Seeds*) in the rest of the graph.

If the subgraphs (slices) should be processed recursively by RECURSIVE OBF [7, 8] they first need to be split into rooted chunks. For the pseudo-code of the algorithm see Algorithm 3. The recursion stops when on a subgraph that is made of a single strongly connected component.

3 CUDA Architecture

The Compute Unified Device Architectures (CUDA) [13], developed by NVIDIA, is parallel programming model and 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

Algorithm 3 RECURSIVE-OBF(V)

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

Procedure 4 OBF-X($\mathsf{Seeds}, \mathsf{Range}$)

1: $Original_Range \leftarrow |Range|$
2: **while** $Range \neq \emptyset$ **do**
3: $Elim, Reached, Range \leftarrow OWCTY(\mathsf{Seeds}, \mathsf{Range})$
4: All elements of *Eliminated* are trivial SCCs
5: $B \leftarrow BWD(\mathsf{Reached}, \mathsf{Range})$
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: $Seeds \leftarrow FWD_SEEDS(B, \mathsf{Range})$
13: **end if**
14: $Range \leftarrow Range \setminus B$
15: **end while**

large set of local 32-bit registers but no cache. 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 a CPU and a *device* code running on the GPU. The device code is structured into so called *kernels*. A kernel executes the same scalar sequential program in many *data independent parallel threads*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp* executes on the multiprocessors in a lock-step manner. When several warps are scheduled on multiprocessors, memory latencies and pipeline stalls are hidden primarily by switching to another warp.

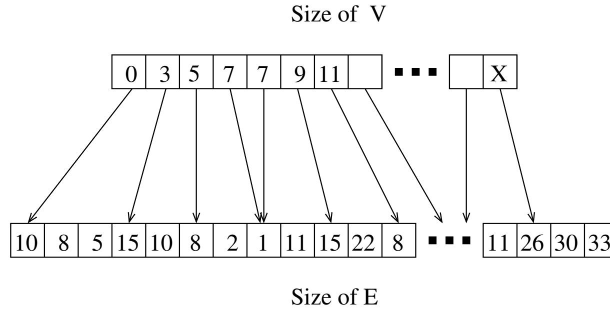

Figure 1: Adjacency list representation

4 CUDA Accelerated SCC Decomposition

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 the adjacency list that is represented as two one-dimensional arrays, similarly as in [18]. One array stores the target vertices of edges sorted according to source vertex. The second array keeps an index to the first array for each vertex. The index points to the first edge emanating from the vertex. See Figure 1. Other data structures needed are organized in vectors, which is a representation that is compatible with CUDA processing. Content stored in these vectors is described together with particular algorithms.

4.1 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, the vector keeps ones only for the vertex/vertices of which the closure should be computed. 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

Algorithm 5 F-KERNEL(G, *visited*, *terminate*)

For all $v \in V$:

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

Algorithm 6 B-KERNEL′(G, *visited*, *terminate*)

For all $v \in V$:

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

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 original vertex. In the rest of the paper we will use an equivalence relation $\sim$ to denote that two vetices are part of the same subgraph, and $[x]$ to denote vertices equivalent to a vertex x, i.e. $[x] = \{v \in V \mid x \sim v\}$.

To compute the backward closure there are two options. 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 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 stick to it. 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

Algorithm 7 FWD-REACH - host code

In: $G = (V, E),\ P \subseteq V$
Out: $\forall v \in V : visited[v] = \texttt{true} \Leftrightarrow \exists u \in P :\ (u, v) \in E^*$

1: **for all** $u \in P$ **do**
2: $\quad$ *visited*[*u*] ← `true`
3: **end for**
4: *terminate* ← `false`
5: **while** *terminate* = `false` **do**
6: $\quad$ *terminate* ← `true`
7: $\quad$ F-KERNEL(V, *visited*, *terminate*)
8: **end while**

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 [19] 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 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.

4.2 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.

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 pseudo-code for FB algorithm listed in Algorithm 9. 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.

4.3 Trimming and self-loop detection

As explained in Section 2, 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.

4.4 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

Algorithm 8 TRIM-KERNEL(G, *eliminated*, *terminate*)

For all $v \in V$:

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

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. 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 $|V|\mathit{size}(v)$ additional space, where $\mathit{size}(v)$ 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

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
```

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 to 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.

4.5 Algorithms

4.5.1 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* indicating which of the vertices belong to the forward respectively backward closure or not (*visited.f*, *visited.b*), vector *elim* keeping the eliminated vertices and vector *pivots* determing the particular pivots for next iteration of the algorithm. Finally the vector *range* identifies the subgraph each vertex belongs to. The UPDATE kernel recomputes the *range* vector (the relation ~) according to the vectors *visited* and *elim*. Moreover it sets the variable *terminate* to the value `true` in the case

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: $map[v],\ inner \leftarrow map[u]$, `true`
5: **end if**
6: **end for**
7: **if** $map[v] = v$ **then**
8: $pivots[v] \leftarrow$ `true`
9: **end if**

that all vertices from previous iteration were visited during the forward and backward reachability or were eliminated.

4.5.2 Coloring algorithm on CUDA

Similarly to the FB algorithm, also the COLORING algorithm can be reformulated as iterative algorithm. Then, 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 FB algorithm 9. Note that the color propagation procedure also computes the vertex (pivot) that the succeeding backward closure is computed from, and that variable *inner* is used to detect that no fix-point has been reached yet.

4.5.3 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 by which phase it is currently processed. See Algorithms 11, and 12.

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 terminating phase. An update procedure is then executed according to the terminating phase.

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] ∈ O then
 6:       UPDATE_O(i)
 7:    end if
 8:    if phase[i] ∈ F then
 9:       UPDATE_F(i)
10:    end if
11:    if phase[i] ∈ B then
12:       UPDATE_B(i)
13:    end if
14:    UPDATE(range, elim, terminate)
15: end while
```

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

For all $v \in V$:

```
 1: if phase[v] = O ∧ reach.o[v] = true then
 2:    if (∀u ∈ V.(u, v) ∈ E ∧ u ~ v) then
 3:       for all w ∈ V. (v, w) ∈ E ∧ v ~ w do
 4:          reach.o[w] = true
 5:       end for
 6:       elim[v] = true
 7:    end if
 8: else
 9:    if phase[v] = B then
10:       for all u ∈ V. (u, v) ∈ E ∧ u ~ v do
11:          reach.b[u] = true
12:       end for
13:    else
14:       for all u ∈ V. (v, u) ∈ E ∧ u ~ v do
15:          reach.f[u] = true
16:       end for
17:    end if
18: end if
```

- 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 a 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 task 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 *half-warp* must at one time perform the same instruction. This is of 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 certain vertices (SCC-closed subsets of one of O, B, F respecting $\sim$) of this process later to finish their decomposition using COLORING algorithm. The sets are chosen if they have less than certain threshold of vertices. Hence we have to compute the sizes and when the regular OBF algorithm ends we also have to finish the decomposition. Despite the inevitable overhead adding this computation is often boosting the algorithm as a whole, as observable from the experiments. Furthermore, we can augment the OBF by prepending one call of trimming to it, in order to skip the costly OBF computation on some trivial components.

5 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 [10] 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. [8], and we applied it to shared-memory environment. For the shared-memory message passing we used lock-free FIFO data structures, as suggested in [4]. Unfortunately, none of our implementations were able to outperform Tarjan algorithm using quad-core architecture, which can be explain by extremely cache-efficient representation of the graph used for Tarjan algorithm that was used on relatively small graphs (we have experimented with graphs of which representation fitted 1 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 280 GPU with 1GB 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 [11], Erdös-Rényi random graph generator; and graphs as produced by enumerative model checker DiVinE [6].

We provide comparison of performance of the following algorithms: serial CPU-based forward reachability, CUDA-based forward reachability, Tarjan's algorithm, CUDA-based FB algorithm (+ trimming), CUDA-based COLORING algorithm, and CUDA-based OBF algorithm (+ trimming, + coloring, + trimming and coloring). Table 1 lists run-times of the algorithms if executed on three types of synthetic graphs with average degree set to twelve and scaled up by the number of vertices. The run-times are also plotted in Figures 2, 3 and 4 using the best time available among versions

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 BFS	446	1135	1915	2712	3563	4440	5331	6479	26021
	GPU REACH	39	80	122	163	205	247	289	356	1501
	Tarjan's	957	2825	3722	5195	6822	8443	10265	12169	50398
	FB	80	180	301	387	511	686	873	962	3980
	FB + Trim	74	156	238	320	402	484	566	648	2888
	Coloring	136	282	427	574	720	866	1041	1159	5205
	OBF	117	269	393	559	723	911	1149	1356	5447
	OBF + Col	124	278	427	620	815	1007	1367	1636	6274
	OBF + Trim	175	309	500	631	793	954	1116	1276	5754
	OBF + Col + Trim	149	312	475	638	802	966	1129	1292	5763
		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 BFS	280	744	1484	1910	3060	3504	3921	4428	14903
	GPU REACH	47	97	124	201	222	254	298	408	1651
	Tarjan's	785	1851	3230	4332	6171	7365	8529	9738	42001
	FB	-	-	-	-	-	-	-	-	-
	FB + Trim	87	206	288	390	524	605	664	814	3578
	Coloring	206	425	527	876	1072	1283	1304	1803	7496
	OBF	-	-	-	-	-	-	-	-	-
	OBF + Col	-	-	-	-	-	-	-	-	-
	OBF + Trim	172	408	545	845	948	1118	1323	1761	7120
	OBF + Col + Trim	175	415	562	851	965	1129	1343	1750	7190
		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 BFS	350	790	1274	1794	2319	2866	3451	4141	16985
	GPU REACH	72	166	307	368	580	671	767	992	3923
	Tarjan's	601	1313	2116	2973	3721	4565	5513	6377	27179
	FB	396	1704	3234	6626	10402	13709	18811	24010	78892
	FB + Trim	195	369	749	901	1446	1666	2073	2846	10245
	Coloring	2344	4970	9300	13354	19908	22055	24735	38337	135003
	OBF	525	1623	3544	5473	8606	12030	16143	20096	68040
	OBF + Col	690	2324	5185	8128	12832	18193	24679	30515	102546
	OBF + Trim	288	660	1246	1512	2118	2652	3593	4556	16625
	OBF + Col + Trim	269	614	1064	1470	1931	2387	3331	4120	15186

Table 1: Run-times for synthetic graphs in milliseconds.

of individual parallel algorithms. Table 2 gives run-times of particular algorithms for graphs corresponding to model checking problems, (also plotted in Figure 5). Note that run-times reported exhibit similar values to the values reported in [18] and [19].

We have observed that the performance of CUDA-based algorithms deeply depend 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 4.1. Generally, we observe that the scalability and efficiency of the parallel reachability procedure effectively limits scalability and

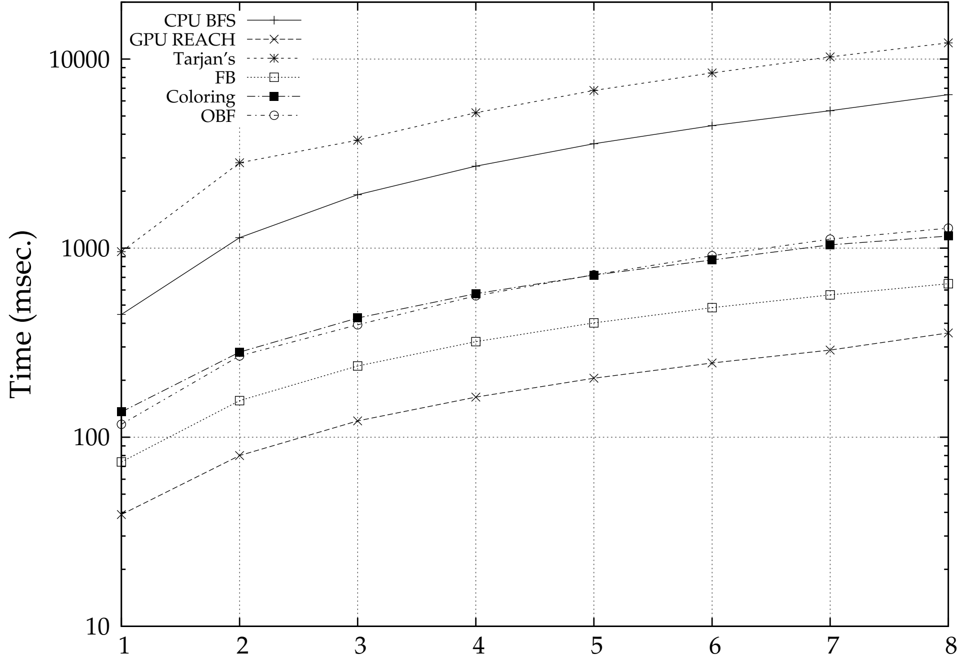

Figure 2: Run-times for Random graphs in milliseconds.

Model (n, m, Number of SCC components)	Algorithm									
	CPU BFS	GPU REACH	Tarjan's	FB	FB + Trim	Coloring	OBF	OBF + Col	OBF + Trim	OBF + Col + Trim
leader-2 (0.7M, 3.8M, 0.7M)	23	13	197	971	85	6	33	34	56	56
phils (0.7M, 6.0M, 59K)	29	8	287	40520	66	104	96	67	97	68
fisher (2.5M, 13.8M, 81K)	84	19	597	34370	128	202	328	177	284	127
anderson-2 (3.1M, 13.4M, 1.6M)	100	43	774	-	171	909	229	256	177	181
leader-1 (3.6M, 26.6M, 3.6M)	129	61	582	21891	627	41	598	149	618	833
elevator-2 (6.4M, 83.3M, 1)	323	84	2437	379	380	5266	485	494	563	570
anderson-1 (8.9M, 47.7M, 4.3M)	325	119	2738	-	1802	3867	915	957	776	866
peterson (9.5M, 42.0M, 18K)	387	79	2740	21891	358	2797	455	511	565	622
elevator-1 (8.6M, 89.4M, 2.0M)	400	100	2933	-	1571	9960	1465	1506	1208	1232
Total	1800	526	13285	-	5188	23152	4604	4151	4344	4555

Table 2: Run-times for model checking graphs in milliseconds.

efficiency of SCC decomposition algorithms. 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 (17 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 over-

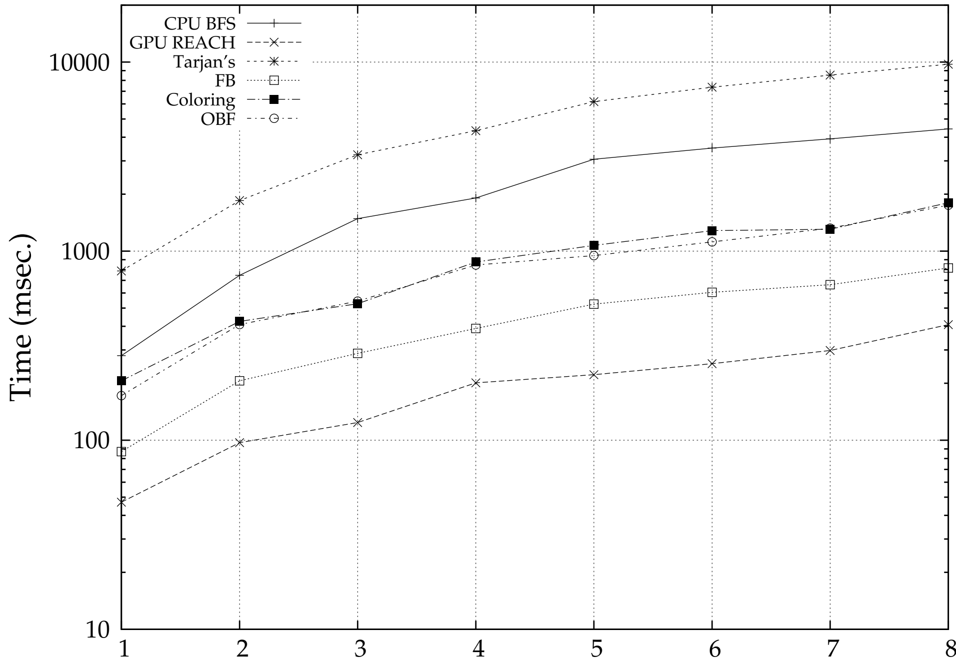

Figure 3: Run-times for R-MAT graphs in milliseconds.

all twelvefold 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 threefold 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 threefold 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 significantly outperforms 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.

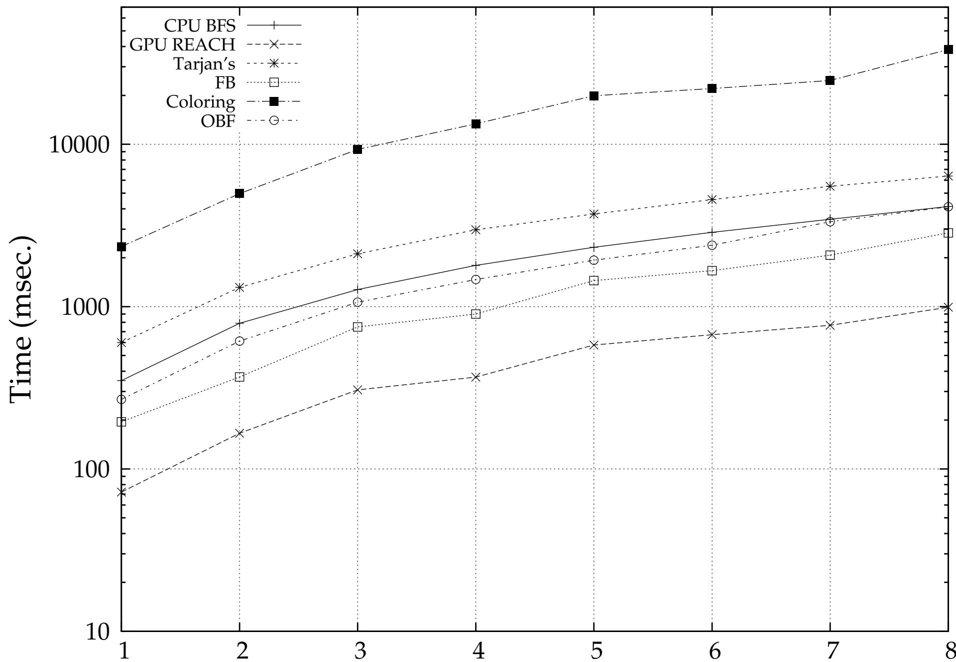

Figure 4: Run-times for SCCA#2 graphs in milliseconds.

6 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.

We have also done an extensive experimental evaluation of known algorithms on several types of graphs proving that with single GTX 280 GPU card we can easily outperform the optimal serial algorithm. However, there is no clear winner among the particular parallel algorithms. While both the COLORING and FB algorithms have the upper hand on some types of graphs, the OBF algorithm seems to fall behind. 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.

In the future we would like to implement parallel algorithms on multiple CUDA devices, for which we already have some initial thoughts, and possibly improve run-

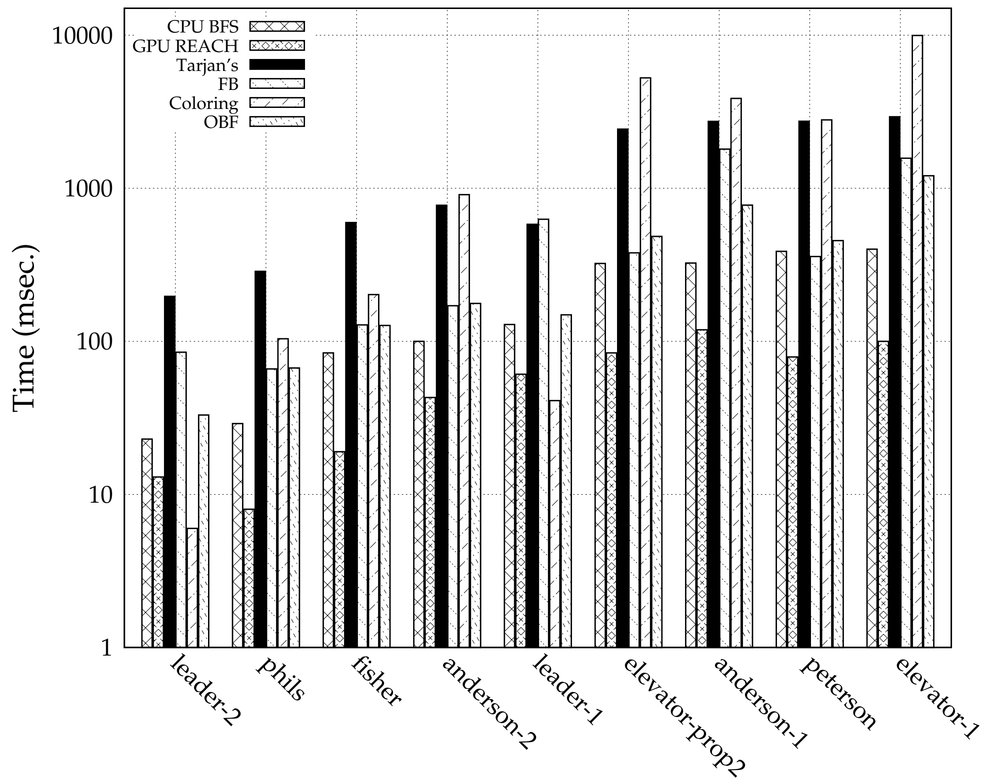

Figure 5: Run-times for model checking graphs in milliseconds.

time by employing the new hierarchical memory of the upcoming generation of CUDA cards [14].

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, 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] Boost: C++ libraries. `http://www.boost.org/`, March 2010.

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

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

[13] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 2.0,. `http://www.nvidia.com/object/cuda_develop.html`, March 2010.

[14] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.co.uk/object/fermi_architecture_uk.html`, March 2010.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

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