Journal of Systems Architecture 60 (2014) 1–10

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Efficient decomposition of strongly connected components on GPUs

Guohui Li [a], Zhe Zhu [a], Zhang Cong [b], Fumin Yang [a,*]

[a] School of Computer Science & Technology, Huazhong University of Science & Technology, China
[b] School of Mathematics & Computer Science, Wuhan Polytechnic University, China

ARTICLE INFO

Article history:
Received 30 January 2013
Received in revised form 12 September 2013
Accepted 18 October 2013
Available online 28 October 2013

Keywords:
SCC decomposition
GPU
Power efficiency
Parallel algorithms

ABSTRACT

The GPU (Graphics Processing Unit) has recently become one of the most power efficient processors in embedded and many other environments, and has been integrated into more and more SoCs (System on Chip). Thus modern GPUs play a very important role in power aware computing. Strongly Connected Component (SCC) decomposition is a fundamental graph algorithm which has wide applications in model checking, electronic design automation, social network analysis and other fields. GPUs have been shown to have great potential in accelerating many types of computations including graph algorithms. Recent work have demonstrated the plausibility of GPU SCC decomposition, but the implementation is inefficient due to insufficient consideration of the distinguishing GPU programming model, which leads to poor performance on irregular and sparse graphs.

This paper presents a new GPU SCC decomposition algorithm that focuses on full utilization of the contemporary embedded and desktop GPU architecture. In particular, a subgraph numbering scheme is proposed to facilitate the safe and efficient management of the subgraph IDs and to serve as the basis of efficient source selection. Furthermore, we adopt a multi-source partition procedure that greatly reduces the recursion depth and use a vertex labeling approach that can highly optimize the GPU memory access. The evaluation results show that the proposed approach achieves up to 41× speedup over Tarjan's algorithm, one of the most efficient sequential SCC decomposition algorithms, and up to 3.8× speedup over the previous GPU algorithms.

1. Introduction

Decomposing a directed graph into its strongly connected components (SCCs) is a fundamental operation in graph algorithms, which is widely used in many different fields such as model checking, computer-aided design and social network analysis. Sequential algorithms of SCC decomposition have been well studied over the past decades. Some well-known solutions, such as Tarjan's [6], Kosaraju's [10] and Dijkstra's [7], achieve linear time complexity. Tarjan's algorithm is often considered the standard and one of the most efficient sequential SCC decomposition algorithms as it traverses each vertex and edge of the graph exactly once.

Limited by manufacturing process and power consumption, the major processor manufacturers have run out of room with most of their traditional approaches to boosting the hardware performance such as clock speed increase, pipelining, branch prediction, out-of-order execution, etc. Therefore, contemporary processors are evolving toward multi-core and many-core architecture while maintaining power efficiency. CPUs with four cores are now common in computers and even in mobile phones and many embedded environments such as set-top boxes and in-car entertainment systems. Modern GPUs push this trend to an extreme, featuring hundreds of cores inside, and delivering a computational throughput over 1 TFlop/s. Although born as graphics processors, GPUs are now widely used in many types of general-purpose computations from embedded systems to supercomputers due to the fact that modern GPUs are designed for optimal performance per watt and thus much more power efficient than CPUs.

In embedded systems, power consumption is often very limited, so it is crucial that the computation in such systems be highly power efficient. The high-performance power-aware GPU architecture is well suited for such computation, and in particular, the GPU and CPU are recently being integrated into a single chip on more and more SoCs which makes the GPU computing widely applicable. Researchers have been seeking to shift many tasks to the GPU and achieved both higher performance and lower power consumption [25,26].

In order to benefit from the new hardware, software needs to be modified or even redesigned to incorporate multi-threaded execution. The transformation from a sequential algorithm to a parallel one is generally not easy, as fundamental changes in the basic idea and underlying data structures are often required. Furthermore, parallel programming in practice is much more difficult than sequential programming because the order in which instructions in different threads are executed is unpredictable and thus extra

* Corresponding author. Tel.: +86 13607137953.
E-mail address: yangfm@hust.edu.cn (F. Yang).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.10.014

care should be taken to consider all the possible results that a program might exhibit. In addition, parallel programming often makes use of locks and barriers to prevent data races, causing high synchronization overhead. Hence, if poorly designed, parallel programs may not run any faster than traditional sequential programs.

Researchers have relied on some architectural features to reduce the synchronization overhead. The most effective approach is to take advantage of atomic operations, which can eliminate data races at a low cost but it requires more consideration because they do not provide mutual exclusion for a segment of code as locks do. Another mechanism that is gaining more and more attention is hardware transactional memory, which allows a group of load and store instructions to execute in an atomic way. However, few processors have implemented this mechanism so far.

GPU programming is generally more complicated than CPU programming. Despite their high memory and computational throughput, it is fairly easy to underutilize the GPU horsepower as GPU programming has many restrictions and requires much more attention from the programmers. Unlike the CPU multithreaded execution which typically uses MIMD, GPUs often make use of a hybrid execution mode of SIMD and MIMD, and the wide SIMD execution is very sensitive to branch divergence, load imbalance and irregular memory access, which can lead to a significant performance hit. In particular, parallel graph algorithms tend to suffer a lot from these issues due to the substantial diversity of graph structure. Hence they are considered to be a class of problems for which it is hard to obtain significantly better performance from GPU parallelization. Moreover, GPUs rely on massively parallel execution for hiding memory latency, typically employing tens of thousands of threads to run concurrently, which is much more than that of CPUs. Therefore, atomic operations will cause much higher overhead because all the operations on the same location will be serialized. As a result, atomic operations should be minimized in GPU programming.

SCC decomposition is particularly difficult to parallelize because generally the sequential SCC decomposition algorithms with linear time complexity all make use of the depth-first search (DFS), which is inherently sequential [8]. Therefore, to benefit from the GPU architecture, completely different approaches need to be applied. Prior work has proposed several CPU parallel algorithms on SCC decomposition. They generally rely on divide-and-conquer approaches and breadth-first search (BFS) to recursively partition the graph into small subgraphs, both of which are relatively suitable for parallelization. Due to the architectural difference between the GPU and CPU, adapting the existed parallel algorithms for GPUs is challenging. To the best of our knowledge, there is only one published work on GPU SCC decomposition [9]. Barnat et al. [9] select three CPU algorithms and introduced the modified GPU versions. However, their implementations are more like a direct transformation from the CPU to GPU, which fails to consider many of the distinguishing features of the GPU such as SIMD execution within a warp and global memory overhead, and thus resulting in significantly underutilization of the GPU capability.

In this paper we present a new GPU algorithm on SCC decomposition which is highly architecture-aware. It uses an iterative variant of the divide-and-conquer approach to partition the graph iteratively. In particular, we use an efficient parallel BFS operation which has linear work complexity to compute the reachability closure. Furthermore, we introduce a multi-source partition procedure that can greatly reduce the recursion depth, which is crucial to good performance of a divide-and-conquer approach. We also design a subgraph numbering scheme, which helps manage the IDs of every subgraph in a safe and efficient way without using atomic operations, and serves as the basis of efficient source selection. In addition, we propose a vertex labeling approach that couples the reachability test, subgraph ID update and SCC output into a single process and further reduces the GPU memory access overhead. The whole implementation is highly optimized for GPUs and aims to fully utilize the GPU horsepower. The evaluation results show that the proposed approach achieves up to 41× speedup over Tarjan's algorithm, one of the most efficient sequential SCC decomposition algorithms, and up to 3.8× speedup over previous GPU algorithms.

The remainder of the paper is organized as follows. We give a brief overview of the basic definitions, modern GPU architecture and prior work on parallel SCC decomposition in Section 2. We present our design and implementation of the GPU SCC decomposition algorithm in Section 3. Experimental results are discussed in Section 4, and finally, we conclude this paper in Section 5.

2. Background and related work

Parallel SCC decomposition is a challenging problem, and doing it on the GPU is even trickier. Yet, some related research exists that acts as the basis of our work. In this section, we first introduce some basic graph definitions and a few concepts in GPU programming. Then we review some previous parallel SCC decomposition algorithms on both the CPU and GPU. In particular, we survey the previous work on GPU BFS, one of the most important basic operations in parallel SCC decomposition algorithms.

2.1. Basic definitions

A directed graph G is denoted as $G = (V, E)$, where V is a set of vertices, and $E \subseteq V \times V$ is a set of directed edges. A *transpose* of a directed graph is the same graph with the edges reversed. If there is a sequence of directed edges $(u, x_1), (x_1, x_2), \ldots, (x_k, v)$ from vertex u to vertex v, we say that v is reachable (or forward-reachable) from u, and that u is backward-reachable from v. We consider a vertex to be reachable from itself. A set of vertices is *strongly connected* if for any two vertices u and v in the set, v is reachable from u. A *strongly connected component* (SCC) is a maximal strongly connected set. In addition, an SCC is *trivial* if it is made of a single vertex c, and $(c, c) \notin E$, and is *non-trivial* otherwise.

We also define two procedures, FWD and BWD, which compute the *forward closure* and *backward closure,* respectively. That is, for $P \subseteq V$, FWD (P, V) computes the vertices in V that are reachable from any vertex in P, and similarly BWD (P, V) computes the vertices in V that are backward-reachable from any vertex in P.

2.2. Modern GPU architecture

In order to deliver high computational throughput, modern GPUs generally adopt two types of parallel execution mode. Within a *warp*, programs are executed in an SIMD mode. That is, each thread executes the same instructions synchronously. Therefore, branch divergence within a *warp* will result in serialization of the different execution paths, thus causing a slowdown. Among *warps*, however, programs are executed in a MIMD mode, and threads are free to diverge.

Threads are grouped into *blocks*. A *block* is a group of threads that will be located on the same multiprocessor and threads in a block can communicate through a local *shared memory*. Threads within a *block* can be explicitly synchronized and are often used for fine-grained parallelization. On the contrary, different blocks are used for coarse-grained parallelization. Global memory access on GPUs can be very expensive if the access made by a warp exhibits low spatial locality. Multiple requests will be serialized if they belong to different cache lines. Hence, special care should be taken to ensure the appropriate memory access pattern.

2.3. Parallel CPU algorithms of SCC decomposition

To the best of our knowledge, the first parallel SCC decomposition algorithm was proposed by Fleischer et al. [1], and was improved later in [3]. Their algorithm is based on two simple lemmas:

Lemma 1. *Let $p \in V$, $F = FWD(p, V)$, then $BWD(p, F)$ is an SCC containing p.*

Lemma 2. *Let $p \in V$, then any SCC is contained in either $FWD(p, V)$ or $V - FWD(p, V)$.*

The proof can be found in [1]. These two lemmas are the basis of nearly all the parallel SCC decomposition algorithms. Note that in Lemma 2, vertex p can actually be extended to a set $P \subseteq V$. When dealing with sparse graphs or highly skewed graphs a multi-source forward closure can partition the graph much more evenly, and thus reduce the recursion depth. This is also the basis of our *Partition* procedure. With this in mind, the algorithm can be easily implemented using a divide-and-conquer approach, which is listed as Algorithm 1. Divide-and-conquer approach is relatively easy to parallelize. As long as the problem is divided into multiple subproblems that are independent of each other, the sub-problems can be solved in parallel. Fleischer et al. also prove the expected serial time complexity to be $O(m \log n)$. Although it is slower than the efficient linear time sequential algorithms, it can outperform the former due to parallel execution. Our GPU algorithm is based on an iterative variant of Algorithm 1, and some GPU-specific optimizations are applied.

Algorithm 1 (*Basic divide-and-conquer SCC decomposition algorithm*).

```
procedure decompose_scc(V)
if V is empty then
    return
p ← random vertex in V
F ← FWD(p, V)
SCC ← BWD(p, F)
output SCC
decompose_scc(F – SCC)
decompose_scc(V – F)
```

Several other parallel algorithms aim to improve the performance of this algorithm for some kinds of graphs. Orzan [2] propose a novel *coloring* algorithm, which gives each vertex a unique color. Whenever the color is updated, it is propagated to vertices with a smaller color. When the colors stabilize, the graph is partitioned into independent subgraphs by each color. Then a backward closure is computed on each vertex keeping its original color, and the result is an SCC. After the SCCs are output, the algorithm is applied to each subgraph recursively. This algorithm works well for graphs with many small SCCs. Barnat et al. [4,5] introduce OBFR-MP, which is reported to perform very well for graphs containing many trivial SCCs. Warren [23] presents a *MultiPivot* algorithm and proved it uses $O(\log^2 n)$ reachability queries. However, most of these algorithms are not well suited for GPUs and more importantly, their performance are not necessarily better than the basic divide-and-conquer algorithm in general.

2.4. GPU algorithms of SCC decomposition

Barnat et al. [9] select three CPU algorithms and introduce modified GPU versions. To our knowledge, this is the only published work that attempted to do SCC decomposition on GPUs. They choose the basic divide-and-conquer approach, the coloring approach and the OBFR-MP approach as the basis and they try to make these three suitable for the GPU environments. However, their implementations are generally not efficient. They use a direct mapping between vertices and threads, which makes most of the threads idle during every iteration as only a small portion of the vertices are actually processed. Some hardware resources are wasted and this poses an overhead to the task scheduler, and for the most important *reachability query* procedure, its complexity becomes $O(n^2 + m)$ instead of $O(n + m)$. Furthermore, assigning one thread to each vertex totally ignores the SIMD processing mode within a warp. The degree of each vertex can be highly irregular, and a thread needs to process each neighbor sequentially. As a consequence, threads within a warp will diverge severely, which results in significant GPU underutilization. If this happens, the CPU implementation will easily outperform the GPU implementation.

2.5. BFS on GPUs

As a basic and one of the most important operations in the SCC decomposition algorithm, the efficiency of BFS can significantly affect the overall performance. In the past few years, BFS algorithms on the GPU have been studied in several works. Harish et al. [11] pioneer the acceleration of BFS on the GPU. However, they use a direct mapping between vertices and threads in a similar way as mentioned in the above section. It results in quadratic work complexity and significant GPU underutilization. Hong et al. [12] use a different mapping strategy. Warps are mapped to vertices rather than threads. Although it still has quadratic work complexity, the warp divergence is largely reduced, and thus it achieves a considerable performance improvement. Luo et al. [13] use a hierarchical technique to implement a queue structure on the GPU, achieving linear work complexity. However, they still use a direct mapping between vertices and threads. Merrill et al. [14] also use a queue structure to perform linear work, and they use a hybrid strategy of fine-grained mapping and warp mapping, delivering significantly higher performance than prior work.

3. SCC decomposition on GPU

3.1. Data representation

Due to the massively parallel processing nature of GPUs, arrays are generally the only kind of efficient data structure we can use when programming on them. Similar to the previous work, we use compressed sparse row (CSR) format to store the graph in GPU main memory. CSR representation contains two arrays, namely column-indices C and row-offsets R as illustrated in Fig. 1. Array C is a concatenation of the adjacency lists of all vertices, whereas each element in array R acts as an index to array C, which indicates the starting point of the adjacency list of that element. CSR representation suits the GPU memory model well because multiple threads can access successive vertices or direct successors of a vertex in parallel in a cache-friendly way and thus it achieves the highest memory throughput.

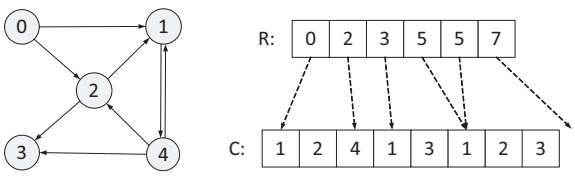


Fig. 1. Example of CSR representation.

In addition to the graph itself, two frontier queues are also used to maintain the vertices when processing a reachability query. A vertex label array is adopted to indicate which subgraph each vertex belongs to as well as which SCC each vertex belongs to. A random number array is applied to store random numbers used in the *Partition* procedure, and we also have some other auxiliary arrays.

3.2. Algorithm overview

Our GPU SCC decomposition algorithm is based on Algorithm 1. However, we need to apply an iterative variant instead as GPUs are not suitable for recursive function calls. The modified version is illustrated in Algorithm 2.

Algorithm 2 (*Main algorithm*).

```
While (TRUE)
  partition the graph into more subgraphs
  empty source
  for each subgraph i in parallel
    source.Enqueue(random vertex in subgraph i)
  if source is empty then
return
    for each v in source in parallel
  ForwardClosure(v)
  update vertex label
  BackwardClosure(v)
  mark SCCs
```

On each iteration, a single source vertex is chosen from each subgraph, and then the forward reachability queries are carried out in parallel. During the query process, reachable vertices form new subgraphs and the corresponding label entries are updated. We then perform the backward reachability queries for these vertices, and the SCCs containing these vertices are identified. Then the identified SCCs are removed and the remaining vertices proceed to the next iteration.

3.3. Reachability query

As we can see from Algorithm 2, reachability query is one of the most important basic operations in the SCC decomposition algorithm. It will be used in several steps of the algorithm. Therefore, the efficiency of the reachability query is crucial to achieve the good performance of the whole process. Furthermore, to fully exploit the GPU horsepower we need to do a reachability query within a subgraph and we need to process multiple reachability queries simultaneously.

We traverse the graph in breath-first order to detect reachable vertices. Due to the independence of the vertices at each BFS level, they can be processed in parallel. An efficient BFS operation optimized for GPUs has been shown to have excellent performance which has the work complexity of $O(m+n)$ [14]. However, like many algorithms, there is a tradeoff between time and space. The linear complexity BFS requires additional $O(m)$ global storage for maintaining the input and output queues. Luckily, modern GPUs typically have large video memory on the graphics cards so this should not be a problem. Our algorithm computes the forward or backward reachability closure during a linear complexity BFS process, and each closure forms a new subgraph.

As for the need to do a reachability query within a subgraph, there are two options to achieve this. For the first option, we can remove edges between the subgraphs, and then the reachability queries can be performed without any knowledge of subgraphs. For the second approach, we can use an *s_label* array of the same size as the vertex number n, and record the subgraph each vertex belongs to. The first approach is clearer as it does not require additional memory and does not hamper the performance of the BFS procedure. However, edge removal is not well suited to be performed in parallel on GPUs, because serial searches will be needed to locate each edge to be removed. We therefore opt for the latter. Furthermore, as shown in Section 3.7, we can combine the subgraph lookup and reachability detection arrays into one array to save memory and processing time.

The forward reachability query algorithm is listed as Algorithm 3. The main structure of the algorithm is similar to the traditional CPU BFS traversal [10] with the queue elements processed in parallel. However, the adjacent list expansion step is highly optimized for GPUs. When dealing with adjacent lists of different sizes, a hybrid strategy of coarse-grained expansion and fine-grained expansion is used to fully utilize the GPU memory and instruction throughput. For backward reachability query, we just do a forward reachability query on the transposed graph.

Algorithm 3 (*Forward reachability query*).

```
procedure ExpandAdjlist(v, list, n, outQueue, grain)
index ← threadID
while index < n
  if grain = coarse_grain
    u ← list[index]
  else
    u ← the vertex ID at address list[index]
  if u not reached yet and s_label[u] = s_label[v]
    mark u as reachable
    outQueue.Enqueue(u)
  index ← index + BLOCK_SIZE
procedure ForwardClosure(inQueue, outQueue, sources)
inQueue.Enqueue(sources)
while inQueue not empty
  empty outQueue
  for each v in inQueue in parallel
    //For large adjacent lists, do coarse-grained expansion
    for each v in the same block having the adjacent list size
  >= BLOCK_SIZE
      list ← adjacent list of v
      ExpandAdjlist(list, list.size, outQueue, coarse_grain)
      adjacent list size ← 0
    //For small adjacent lists, use prefix sum to scatter the
  addresses and then do fine-grained expansion
    scatter the addresses of each remaining v's adjlist entries
  into an idx_list in each block's shared memory
    ExpandAdjlist(v, idx_list, idx_list.size, outQueue, fine_grain)
  inQueue ← outQueue
```

3.4. Graph partitioning

For divide-and-conquer algorithms, one of the key factors to affect the overall performance is the quality of the divide step, *i.e.* each divide step should divide the current problem into two or more sub-problems of similar sizes. That way, the depth of the recursive tree will be $O(\log n)$ and the whole problem can be solved in $O(n\log n)$ time complexity. However, if the sizes of the sub-problems are highly skewed, the recursion depth may degrade to $O(n)$ and the time complexity may become $O(n^2)$.

When we choose a source vertex v in a subgraph, the forward reachability closure divides this subgraph into two smaller subgraphs, namely vertices reachable from v and vertices not

reachable from v. When the graph is sparse or the degree distribution is highly skewed, the size of the forward closure may be very small and this can result in a poor partition. For example, a graph with no edges will have recursion depth n. Our GPU kernel is an iterative variant of the divide-and-conquer approach, which finishes all the work from one level of the recursive tree at a time. That means the times needed to launch the GPU kernel is directly affected by recursion depth. Moreover, the GPU kernel launches come with a cost higher than normal function calls, so the number of kernel launches should be minimized. Fig. 2 plots the recursion depth in log-scale for graphs of different average degree, using random graphs generated by GTgraph [14] with 0.1 million vertices and different number of edges. For the blue curve, the graphs are decomposed using the single-source approach we just presented, which reveals that the recursion depth will increase dramatically as the average degree decreases.

Barnat et al. [9] use a *TRIMMING* procedure to solve this problem, which iteratively removes vertices that have no immediate predecessors or successors because these vertices cannot be part of any non-trivial SCCs. Hence they can be safely removed from the graph as trivial SCCs. But this is only effective for certain kinds of graphs, and the *TRIMMING* procedure is an expensive operation itself. We have, therefore, opted for a different solution.

We add a procedure called *Partition* before each iteration. It works by randomly choosing a certain number of vertices from the whole graph, and do a multi-source forward reachability query. Each source vertex still does the query within its underlying subgraph. After this procedure, a subgraph will be divided into two subgraphs: reachable from those sources and not reachable from them. As the computed forward closures are the union of multiple single-source forward closures, we can no longer do backward reachability queries and determine any SCCs. However, it is guaranteed that all SCCs are constrained in the subgraphs. In the best situation, a graph containing k subgraphs will be partitioned into $2k$ smaller subgraphs, so we can proceed to do the following single source reachability queries.

The problem now lies in how to choose these multiple source vertices. The *Partition* procedure has best quality when half of the remaining vertices are reachable in each subgraph from the source vertices. However, as we do not know the size of the reachability closures a priori, it is difficult for us to efficiently choose the appropriate source vertices. If it has noticeable overhead, the performance gain may be easily outweighed by the penalty for carefully choosing the source vertices. Therefore, we instead use uniform random numbers for vertex permutation and the average degree of the graph is adopted as a heuristic to determine the number of vertices to be chosen.

Parallel random number generation on GPUs will be discussed in Section 3.5. Once we have a random permutation of all the vertices, we choose the first k vertices and remove the SCCs which have already been identified. The remaining vertices are then put into the input queue to initiate the multi-source reachability query. Provided that the random numbers are uniformly distributed in general and the set of generated random numbers on each iteration do not exhibit any correlations, the partition appears to be very effective and has little overhead. The value of k is computed from the average degree d of the graph, which is defined as the ratio of edge number versus vertex number. By default, let $k = \min(n/10, n/d^2)$. According to our experience it performs well in most cases. In addition, we can also adjust the value of k as a command-line argument. The red curve in Fig. 2 shows that our *Partition* procedure can greatly reduce the recursion depth. Fig. 3 presents recursion depth for different value of k using random graphs with 0.1 million vertices and 0.1 million edges. We can see that when the value of k reaches a certain threshold, increasing it more won't result in much benefit.

3.5. Source selection

Both the partition and the single-source reachability query steps require random source vertex selection. The difference is that for the former, multiple vertices from the whole graph are selected, whereas for the latter, we need to select a single vertex in each subgraph. The requirements for the statistical quality of the random numbers are different, too. As mentioned in the previous section, the *Partition* procedure needs generally uniformly distributed random numbers and independent sets of them between iterations to increase the partition quality, because if the numbers have certain obvious patterns, we might end up doing reachability queries on the same sets of vertices and fail to divide the graph into more subgraphs. On the contrary, single-source reachability query procedure is not very sensitive to the vertices selected in each subgraph, as after each backward reachability query, SCCs containing those vertices will be removed and new vertices will be selected on the next iteration. For graphs that have certain patterns between vertex IDs and edge distribution, performance may suffer from poor randomness of the selected vertices. In that case, we can randomly permute the vertex IDs beforehand on the CPU as an offline preprocessing step.

Research on random number generators suitable for GPUs is still in its infancy. Due to the requirement of generating a massive amount of random numbers simultaneously and the limits of the GPU programming model, many high-quality algorithms on CPUs are not applicable [21,22]. Some simple algorithms that are highly

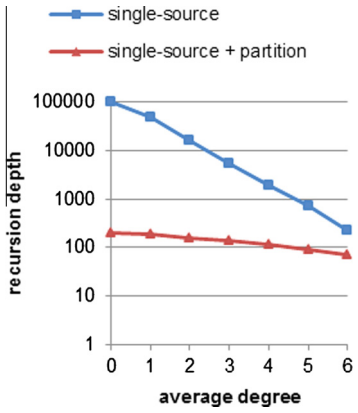


Fig. 2. Recursion depth for different average degree. Random graphs with 0.1 M vertices.

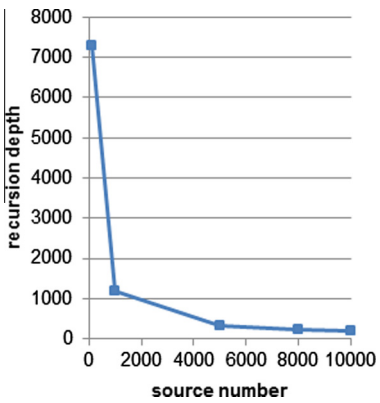


Fig. 3. Recursion depth for different source number. Random graphs with 0.1 M vertices and 0.1 M edges.

efficient but do not have very good statistical properties however, are well suited for GPUs. In our case, the *Partition* procedure only needs basic randomness. Whether or not the generator can pass certain tests is not an issue. Therefore, we try to improve the performance in random number generation.

We use the well-known linear congruential generator, which has the simplest form and best performance:

$$X_{n+1} = ax_n + c(\mathrm{mod}m). \quad (1)$$

We use a = 1664525 and c = 1013904223, which is suggested in [24], and let m be the number of vertices of the graph. In addition, we use an array of the same size as the vertex number to record the last set of random numbers. At the initialization stage, the array is filled with CPU generated random numbers as x_0. On each iteration, every thread reads in the last random number x_n and then computes x_{n+1} using Eq. (1), and finally it writes back the value x_{n+1} to seed the next iteration. Furthermore, if the vertex number n is near to some power of two, we can avoid the expensive modulo operations and replace it with efficient bitwise operations.

In the case of selecting a single vertex in each subgraph, we employ a different approach. The basic idea is the same as [9]. We allocate a single location for each subgraph, and have each thread write the vertex ID to the location corresponding to its subgraph ID. According to the GPU programming model, when multiple threads in a warp write to the same location using a non-atomic instruction, only one thread actually performs the write [16]. Consequently, the threads writing to the same location will vie for a winner without causing too much memory overhead, and at the same time we also gain some randomness.

The locations to do these writes are the main challenge here, because we do not know how much memory to allocate and how to map subgraph IDs into the array beforehand. Barnat et al. [9] encounter the same problem, and they introduce two solutions. For the first solution, one extra space is allocated for every vertex. After a source vertex partition its underlying subgraph into some subgraphs by reachability queries, each subgraph uses that source vertex's extra space as the location to do the election writes. However, the subgraphs are serialized for the usage of that location, resulting in a performance penalty and many unused spaces. For the second solution, an array of a certain size is allocated, and each subgraph is given a unique subgraph ID according to a complete ternary tree. The subgraph IDs are used as indices to the array. This solution also suffers from many unused spaces in the array, and what's worse, the subgraph IDs would soon grow beyond the size of the array, and thus requiring a renumbering process.

As we can see, to solve this problem more efficiently, we need a new way to manage the subgraph IDs. We introduce a subgraph numbering scheme in Section 3.6. With this approach, we can solve this problem in a highly efficient and clear way.

3.6. Subgraph numbering scheme

As the algorithm proceeds, the number of subgraphs will increase in an irregular way that cannot be predicted in advance, and the parallel execution of thousands of threads makes it worse. As a consequence, giving every subgraph an appropriate ID is a challenge. Generally, for the single-source selection and reachability query to work efficiently, the subgraph numbering needs to meet the following two requirements:

1. Subgraph numbering should be as compact as possible. In other words, from a global point of view, subgraph IDs should increase progressively from zero, and one at a time without skipping any value. Because of this, we do not need to worry about the subgraph IDs growing beyond the size of the array, and the overhead of re-initializing the array before each iteration is minimized. Unfortunately, when the *Partition* procedure is added, totally continuous subgraph IDs is just impossible, because occasionally a new subgraph would completely overlap the old one.
2. Each subgraph should know the new subgraph ID assigned to it when it is partitioned into two smaller subgraphs before the reachability query starts. This makes all subgraphs agree on the global numbering scheme, and each vertex can easily update its label when it is visited during the traversal.

Due to the massively parallel execution of GPUs, we need to be extremely careful when dealing with shared variables. Using atomic operations to increase the subgraph IDs seems like a safe and reasonable way. However, atomic operations are expensive and moreover, all operations on the same location are serialized. In general, they do not scale to thousands of threads and their use should be minimized. As we can see, with enough care, the use of atomic operations can be avoided altogether.

Our subgraph ID numbering scheme is depicted in Fig. 4. We maintain a variable *max_subgrah* to denote the maximum number of subgraphs the whole graph have so far and it is initialized to 1. Note that this value may be larger than the actual number of subgraphs which currently exists, because some subgraphs may have disappeared due to overlapping or SCC removal. We use a source election array for single-source selection, an entry of which represents a subgraph, and it is indexed by subgraph IDs. Before each iteration, the source election array is initialized to -1 for the first *max_subgrah* elements. Each vertex writes its vertex ID to the location corresponding to its underlying subgraph ID. Then we do a parallel prefix sum [22,27] on these *max_subgrah* elements to compute the index of each element that is not -1, and scatter them to the input queue for reachability query. The total sum N of the prefix sum operation is the number of new subgraphs that will be formed. So on the next iteration the value of *max_subgraph* will be updated to *max_subgraph* + N.

Now we need to assign a new unique subgraph ID to each subgraph, which will be used when the new subgraph is formed inside each subgraph. A *subgraphID_update* array is used to record this information. With the knowledge of the upper bound of the total number of subgraphs by the end of this iteration, the subgraph ID assignment is actually straightforward. We just employ N threads to process the elements in the input queue. Each thread reads in one vertex, looks up its corresponding *subgraphID*, and stores the value *max_subgrah* + *threadID* to the location *subgraphID_update*[*subgraphID*]. The single-source selection and subgraph numbering algorithm is listed as Algorithm 4.

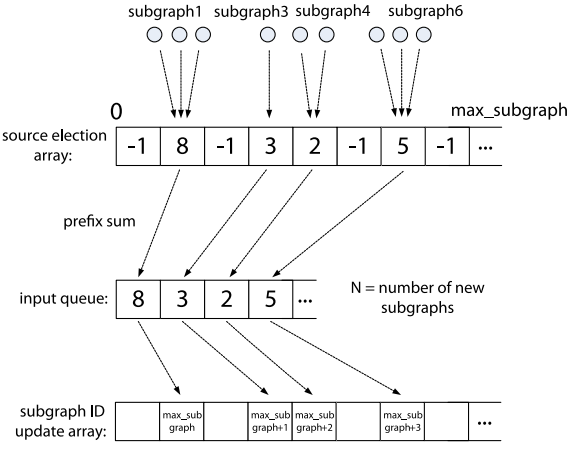


Fig. 4. Example of the subgraph numbering scheme.

The subgraph numbering in the *Partition* step is similar to the single-source selection step. The difference is that we only have every vertex which is randomly chosen write its vertex ID to the source election array rather than having all remaining vertices write their vertex IDs to the corresponding location.

Algorithm 4 (*Single-source selection and subgraph numbering algorithm*).

Function: PrefixSum($rank_i$) performs a parallel prefix sum where each thread t_i is returned the sum of $rank_0$ to $rank_i$ and the total sum
procedure SelSrc(*s_label*, *source_election*, *subgraphID_update*, *inQueue*, *max_subgraph*)
for *i* = 0 to *max_subgraph* - 1 **in parallel**
source_election[*i*] ← -1
foreach *v* in the remaining graph vertices **in parallel**
subgraphID ← *s_label*[*v*]
source_election[*subgraphID*] ← *v*
for *i* = 0 to *max_subgraph* - 1 **in parallel**
rank ← *source_election*[*i*] = -1 ? 0 : 1
(*offset*, *N*) ← PrefixSum(*rank*)
if *source_election*[*i*] != -1 **then**
inQueue[*offset*] ←*source_election*[*i*]
for *i* = 0 to *N* - 1 **in parallel**
subgraphID ← *s_label* [*inQueue*[*i*]]
subgraphID_update[*subgraphID*] ← *max_subgraph* + *threadID*
max_subgraph ← *max_subgraph* + *N*

3.7. Vertex labeling

Reachability status update, subgraph attachment update and SCC output all need vertex labeling. Consequently, the efficiency of the labeling process has an impact on the overall performance. We show a vertex labeling scheme highly optimized for GPUs.

3.7.1. Coupling of reachability label and subgraph label

A straightforward implementation would use two arrays *r_label* and *s_label* to maintain reachability status and subgraph attachment, respectively. During a reachability query process, reachable vertices are marked in *r_label*. When the query finishes, *s_label* is updated according to new assigned subgraph IDs and information in *r_label*. Then the *r_label* needs to be re-initialized for the next procedure. This however has a relatively high cost because after each reachability query the *s_label* update and *r_label* re-initialization may cause up to $2n$ memory writes and $2n$ memory reads, and additional array takes up more memory, too. We use an approach coupling *r_label* and *s_label* into one array, and thus eliminate the additional memory access.

To achieve this, we need to modify the *subgraphID_update* array to hold the subgraph ID increments instead of new subgraph IDs. In this way we can update the subgraph ID of each vertex on-the-fly when traversing the graph and still be able to obtain the value of the subgraph ID before update. To be exact, we apply the following operation to set up *subgraphID_update* array:

$$subgraphID\_update[max\_subgrah + threadID] = max\_subgrah + threadID - subgraphID.$$

3.7.2. SCC output

SCCs are identified when a backward reachability query is finished. Because every subgraph has a unique subgraph ID starting from 0, the SCC labeling is rather easy. We just negate the corresponding subgraph ID when a vertex is reached and then it can be used as SCC IDs.

The whole vertex labeling algorithm is illustrated in Algorithm 5. Note that both reachability test and subgraph update are performed, and we only use the *s_label* array, which is initialized to 0. Furthermore, no re-initialization is needed during the whole process.

Algorithm 5 (*Vertex labeling*).

procedure LabelVertex(*vertexID, predecessorID, s_label, subgraphID_update*)
subgraphID ← *s_label*[*vertexID*]
pred_subgraphID ← *s_label*[*predecessorID*]
if forward traversal **then**
subgraphID_increment ← *subgraphID_update*
[*pred_subgraphID*]
if *subgraphID* = *pred_subgraphID* - *subgraphID_increment*
then
s_label[*vertexID*] ← *pred_subgraphID*
Enqueue *vertexID* for subsequent traversal
else
vertexID is already processed or not reachable
else
if *subgraphID* = -1 ∗ *pred_subgraphID* **then**
s_label[*vertexID*] ← *pred_subgraphID*
Enqueue *vertexID* for subsequent traversal
else
vertexID is already processed or not reachable

4. Experimental results

We compare the performance of the proposed algorithm with Tarjan's algorithm and the GPU SCC decomposition algorithms in previous work. Tarjan's algorithm is one of the most efficient sequential SCC decomposition algorithms. It uses a stack and DFS to traverse all the vertices and edges exactly once, and thus it has a time complexity of $O(m + n)$. We implement our own version of Tarjan's algorithm, and make use of five arrays instead of other fancy data structures to record states during the process in order to trade space for time, resulting in an extremely efficient implementation, which outperforms the previous implementation [9] by about two times. Barnat et al. [9] introduces several different SCC algorithms. As we are unable to get the source code of their implementation, we extract the best results reported in their paper.

We use several kinds of graphs for input. Random, R-MAT [18] and SSCA#2 [19] are synthetic graphs generated using GTgraph [15]. The rest are real-world graphs from the University of Florida Sparse Matrix Collection [20]. The details of the graphs are listed in Table 1. Our GPU algorithm are implemented using CUDA 4.2 [17], and all experiments are run on a host machine with 4 GB memory, an Intel 4-core 3.4 GHz Core i7 2600 k CPU and an Nvidia Geforce GTX 480 GPU. Note the GPU is the same with that used by Barnat et al. [9]. Because these algorithms are run entirely on the GPU, other hardware such as the CPU and system memory have negligible impact on the run time. Therefore, the results are comparable.

In addition, we have compared the results of our GPU algorithm with that produced by the CPU algorithm to verify that all the SCCs are correctly decomposed.

The results are presented in Table 2, Fig. 5 and Fig. 6. As we can see, for synthetic graphs, our algorithm performs very well and achieves over 10× speedup for most of the graphs compared to Tarjan's algorithm. In particular, we achieve 41× speedup for random graphs because the edges are distributed uniformly so the GPU algorithm can effectively exploit the parallelism. R-MAT graphs have highly irregular degree distribution with nearly

Table 1
Selected graphs for experiments.

Name	Description	Vertices (10^6)	Edges (10^6)	SCCs
random1	Uniformly random graph	1.0	12.0	16
random2	Uniformly random graph	2.0	24.0	31
random3	Uniformly random graph	3.0	36.0	32
rmat1	R-MAT (A = 0.45, B = 0.15, C = 0.15)	1.0	12.0	0.48M
rmat2	R-MAT (A = 0.45, B = 0.15, C = 0.15)	2.0	24.0	0.97M
rmat3	R-MAT (A = 0.45, B = 0.15, C = 0.15)	3.0	36.0	0.97M
ssca1	SSCA#2	1.0	30.0	576
ssca2	SSCA#2	2.0	60.0	1.1K
ssca3	SSCA#2	3.0	90.0	1.7K
amazon-2008	Book similarity network, Amazon	0.7	5.2	91K
amazon0505	Amazon product co-purchasing network from May 5 2003	0.4	3.4	14K
language	Finite-state machine for natural language processing	0.4	1.2	2.5K
flickr	2005 crawl of flickr.com	0.8	9.8	0.28M
FullChip	Circuit simulation	3.0	26.6	35
pre2	AT&T, harmonic balance method, large example	0.7	6.0	391

Table 2
Run time in milliseconds for different algorithms running on different graphs.

Name	Our algorithm	Recursion Depth	Tarjan's algorithm (speedup)	Barnat's algorithms (speedup)
random1	13	9	432 (33×)	36 (2.8×)
random2	24	17	980 (41×)	73 (3.0×)
random3	38	23	1560 (41×)	111 (2.9×)
rmat1	35	560	295 (8.4×)	36 (1.0×)
rmat2	66	1020	647 (9.8×)	74 (1.1×)
rmat3	102	1440	975 (9.6×)	134 (1.3×)
ssca1	24	86	261 (11×)	72 (3.0×)
ssca2	46	150	583 (13×)	155 (3.4×)
ssca3	74	215	890 (12×)	284 (3.8×)
amazon-2008	31	450	103 (3.3×)	
amazon0505	14	190	72 (5.1×)	
language	5	75	32 (6.4×)	
flickr	54	540	110 (2.0×)	
FullChip	25	35	280 (11×)	
pre2	7	60	57 (8.1×)	

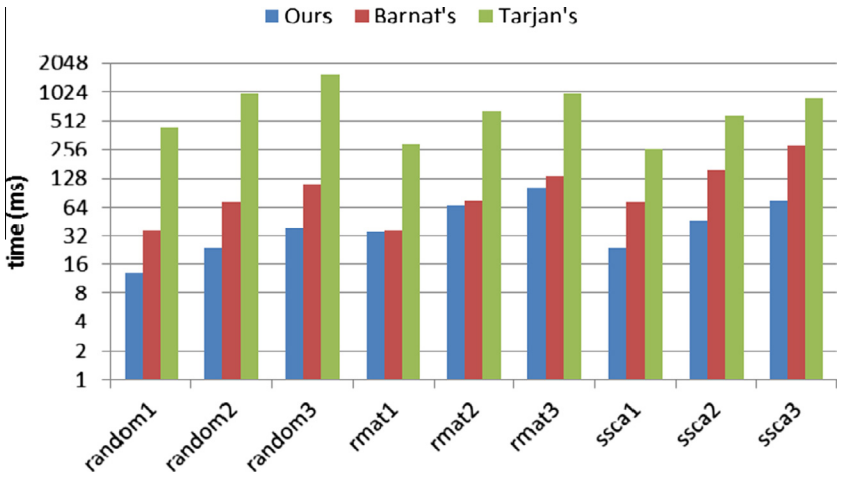


Fig. 5. Run time for different algorithms on synthetic graphs.

half of the vertices having zero degree and many vertices having a degree of several hundred. Thus there are a lot of trivial SCCs in such graphs, and it is difficult for the *Partition* procedure to partition the graph evenly. Therefore, the recursion depth tends to be very high, resulting in a performance penalty, and we achieve up to 9.8× speedup. SSCA#2 graphs typically contain many non-trivial SCCs and little trivial SCCs so the total number of SCCs is small, but they have a large number of edges which benefits the CPU more because adjacency lists exhibit spatial locality and CPUs have a more sophisticated cache system. Nonetheless, we achieve up to 13× speedup. When compared to previous GPU algorithms, our algorithm outperforms them by about three times except for R-MAT graphs. They apply a *TRIMMING* procedure to improve performance, which removes the leading and terminal vertices iteratively as trivial SCCs. R-MAT graphs can greatly benefit from this technique because nearly half of the vertices are trivial SCCs and are all removed on a single iteration of the *TRIMMING* procedure.

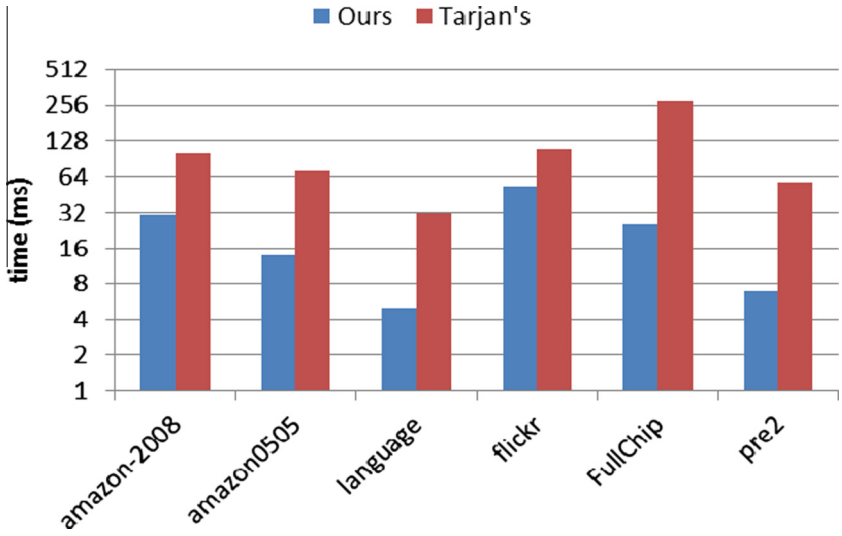


Fig. 6. Run time for different algorithms on real-world graphs.

For real-world graphs, degree distribution is typically much more irregular than synthetic graphs. To test the adaptability of our algorithm, we choose six graphs from different fields. In general, the vertex degree of the selected graphs can range from zero to hundreds of thousands, resulting in poor partition of the graphs and significant load imbalance on GPUs. From the results we can see that, although not as impressive as on synthetic graphs, our algorithm can handle different graph structure in the real world and achieves 2.0× to 11× speedup over Tarjan's algorithm.

The experimental results also reveal that the performance of our GPU algorithm is mainly affected by the recursion depth whereas Tarjan's algorithm is mainly affected by the number of vertices and edges. Thus if the quality of the partition step is further improved, for example by carefully selecting the source vertices or adjusting the source number, the proposed algorithm has potential to obtain even higher performance.

5. Conclusion

Utilizing the high-performance low-power GPU architecture to accelerate traditional applications has become an important approach to power aware computing. We have demonstrated an efficient GPU algorithm for SCC decomposition. Through a divide-and-conquer approach and massively parallel execution, our GPU algorithm outperforms the optimal sequential algorithm significantly for all the graphs we use in our experiments.

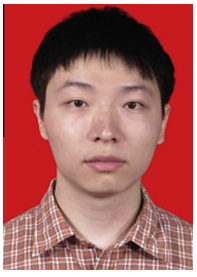
In order to fully exploit the parallelism of the GPU architecture, our algorithm adopts several techniques that are specifically designed for GPUs. In particular, we use an efficient parallel BFS operation which has linear work complexity to compute the reachability closure. And we introduce a partition procedure that can greatly reduce the recursion depth, which is the key to good performance of a divide-and-conquer approach. We have also designed a subgraph numbering scheme, which helps manage the IDs of every subgraph in a safe and efficient way. This numbering approach is critical to source selection process as it determines the locations needed for vertex election. In addition, we propose a vertex labeling approach that couples the reachability test, subgraph ID update and SCC output into a single process and eliminates the use of additional arrays and operations, resulting in further improvement of performance.

References

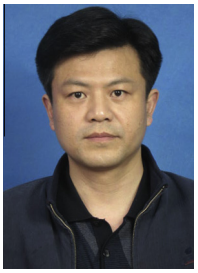
[1] L.K. Fleischer, B. Hendrickson, A. Pinar. On identifying strongly connected components in parallel, in: Parallel and Distributed Processing, vol. 1800 of LNCS, Springer, 2000, pp. 505–511.
[2] S. Orzan. On Distributed Verification and Verified Distribution, PhD thesis, Free University of Amsterdam, 2004.
[3] D. Coppersmith, L. Fleischer, B. Hendrickson, A. Pinar. A divide-and-conquer algorithm for identifying strongly connected components, Technical Report RC23744, IBM Research, 2005.
[4] J. Barnat, J. Chaloupka, J. van de Pol, Distributed algorithms for SCC decomposition, J. Logic Comp. 21 (1) (2011) 23–44.
[5] J. Barnat, P. Moravec, Parallel Algorithms for Finding SCCs in Implicitly Given Graphs, in: Formal Methods: Applications and Technology, vol. 4346 of LNCS, Springer, 2006, pp. 316–330.
[6] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[7] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, NJ, Ch.25, 1976.
[8] J.H. Reif, Depth-first search is inherently sequential, Inf. Process. Lett. 20 (5) (1985) 229–234.
[9] J. Barnat, P. Bauch, L. Brim, M. Ceska. Computing Strongly Connected Components in Parallel on CUDA, in: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11), IEEE Computer Society, 2011, pp. 541–552.
[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press, 2009.
[11] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: HiPC'07: Proceedings of the 14th international conference on High performance computing, Berlin, Heidelberg, Springer–Verlag, 2007, pp. 197–208.
[12] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, in: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP'11, New York, NY, USA, ACM, 2011, pp. 267–276.
[13] L. Luo, M. Wong, W. Hwu, An effective gpu implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference, DAC'10, New York, NY, USA, ACM, 2010, 52–55.
[14] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU Graph Traversal, in: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'12), ACM, 2012, pp. 117–128.
[15] D.A. Bader, K. Madduri, GTgraph: A synthetic graph generator suite in: Technical Report GA 30332, Georgia Institute of Technology, Atlanta, 2006.
[16] NVIDIA *CUDA C Programming Guide Version 4.2*. https://developer.nvidia.com/cuda-downloads, 2012.
[17] NVIDIA, CUDA. Available from: <http://www.nvidia.com/cuda/>.
[18] D. Chakrabarti, Y. Zhan, C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In SDM, SIAM, 2004, pp. 442–446.
[19] D.A. Bader, K. Madduri, Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In HiPC, volume 3769 of LNCS, Springer, 2005, pp. 465–476.
[20] University of Florida Sparse Matrix Collection. Available from: <http://www.cise.ufl.edu/research/sparse/matrices/>.
[21] M. Manssen, M. Weigel, A.K. Hartmann, Random number generators for massively parallel simulations on GPU, Eur. Phys. J. Special Topics 210 (2012) 53–71.
[22] Hubert Nguyen, GPU gems 3, Addison-Wesley Professional, 2007.

[23] S. Warren, Finding Strongly Connected Components in Parallel Using O(log2n) Reachability Queries, In SPAA, ACM, 2008, pp. 146–151.
[24] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd edition., Cambridge University Press, Cambridge, 2007.
[25] K.T.T. Cheng, Y.C. Wang, Using mobile gpu for general-purpose computing – a case study of face recognition on smartphones, International Symposium on VLSI Design, Automation and Test (VLSI-DAT), IEEE, 2011, pp. 1–4.
[26] G. Calandrini, A. Gardel, P. Revenga, J.L. Lázaro, GPU Acceleration on embedded devices a power consumption approach, in: Proceedings of the 14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE, 2012, pp. 1806–1812.
[27] D. Merrill, A. Grimshaw, Parallel Scan for Stream Architectures, Technical Report #CS2009-14, Department of Computer Science, University of Virginia, 2009.

Guohui Li is a full professor in School of Computer Science & Technology, Huazhong University of Science & Technology in China. He got his Phd degree in 1999 in Huazhong University of Science & Technology. His current main research interests include advanced data management and embedded real-time computing. He has published some papers in international journals and conferences with high reputation such as IEEE Transactions on Computers, RTSS, AAAI, Information Systems, The Computer Journal, Journal of Systems and Software and Journal of Systems Architecture.



Zhe Zhu received the bachelor's degree from Huazhong University of Science & Technology in 2006. He is currently a PhD candidate in School of Computer Science & Technology, Huazhong University of Science & Technology. His research interests include GPU computing, parallel programming, embedded application, software engineering and power-efficient architecture.



Zhang Cong is a professor in School of Mathmatic & Computer Science, Wuhan Polytechnic University. He received the Ph.D. degree in Computer Application Technology from Wuhan University in 2010, and the master's degree in Computer Application Technology from Wuhan University of Technology in 1999, and bachelor's degree in Automation Engineering from Huazhong University of Science and Technology in 1993. His research interests include multimedia signal processing, multimedia communication system theory and application, pattern recognition.

Fumin Yang is a full professor in School of Computer Science & Technology, Huazhong University of Science & Technology in China. He got his master's degree in 1990 in Huazhong University of Science & Technology. His current main research interests include database management system (DBMS), database security, parallel DBMS, spatial database and embedded real-time computing. He has published over 50 papers in core journals.