Mapping the MPM maximum flow algorithm on GPUs

Steven Solomon and Parimala Thulasiraman
Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada
E-mail: umsolom9@cs.umanitoba.ca, thulasir@cs.umanitoba.ca

Abstract. The GPU offers a high degree of parallelism and computational power that developers can exploit for general purpose parallel applications. As a result, a significant level of interest has been directed towards GPUs in recent years. Regular applications, however, have traditionally been the focus of work on the GPU. Only very recently has there been a growing number of works exploring the potential of irregular applications on the GPU. We present a work that investigates the feasibility of Malhotra, Pramodh Kumar and Maheshwari’s “MPM” maximum flow algorithm on the GPU that achieves an average speedup of 8 when compared to a sequential CPU implementation.

1. Introduction
Recently, Graphics Processing Units (GPUs) have become powerful co-processors for general purpose parallel applications. As a result of the GPU’s high degree of parallelism and computational power, significant interest has been directed toward the platform for parallel application development. A vast portion of the literature, however, focuses on applications for solving regular problems. Regular problems exhibit structured and predictable data access patterns and typically allow a developer to exploit data-parallelism in an intuitive manner. As a result, regular applications generally map well to the GPU architecture.

Irregular problems, however, have not been as extensively studied on the GPU. Irregular problems use pointer-based data structures such as graphs or trees. Maximum flow computations or unstructured grid problems are two such examples of irregular problems. Due to the composition of the data, these problems exhibit unstructured and unpredictable data access patterns. As a result, irregular problems are significantly more difficult to implement efficiently in a parallel manner. Ruetsch and Micikevicius [1] detail an example of the performance degrading effects of inefficient memory usage on the GPU. Efficiently mapping an irregular problem to the GPU becomes difficult as a result of the unpredictable data access patterns.

We investigate an irregular, graph-based algorithm that solves the maximum flow problem. In general, graph theory applications make use of the graph data structure. A graph is composed of a set of vertices that are connected by edges. These edges can be visualized as a road connecting one intersection (vertex) to another. An edge is said to be directed if it allows movement in only one direction, or undirected if this restriction does not exist. A path between two vertices \( a \) and \( b \) traverses some arbitrary number of edges (starting from \( a \)) until \( b \) is reached. A shortest path between two vertices \( a \) and \( b \) is a path of minimum length from \( a \) and \( b \) (that is, a path containing the minimum number of edge-traversals required to move from \( a \) to \( b \)).
A flow network expands the concept of the graph by introducing a flow capacity for each edge that describes how much flow an edge can support. In network theory, the flow can be visualized as data traversing some link between nodes (vertices) in a network graph.

For our implementation, we focus on the use of general, or random, graphs. A general graph emphasizes the irregular nature of graph data as vertices are connected by edges to other vertices in unstructured, random patterns. This approach differs from the use of more structured grid graphs by removing the possibility of optimizations which take into account the structured nature (as was shown by Hussein et al. [2]).

The maximum flow problem has applications in a variety of areas. For example, Hussein et al. [2] use a maximum flow algorithm to perform image processing (in the form of image segmentation). The maximum flow problem can also be applied to transportation problems (which further have applications in network theory), such as the transshipment problem [3].

In this paper, we consider the mapping and implementation of Malhotra, Pramodh Kumar and Maheshwari’s [4] maximum flow algorithm (MPM algorithm) on NVIDIA GPUs using CUDA. In the literature, Goldberg and Tarjan’s [5] push-relabel maximum flow algorithm has been the target for parallelization on different computer architectures and this is no different for the GPU. Section 3 will discuss some of the current implementations in the literature.

To the best of our knowledge, there exists no current work on parallelizing the MPM algorithm on the GPU. We believe that though the push-relabel algorithm is the best-known sequential maximum flow algorithm, it is not the best suited algorithm for the GPU. The synchronous nature of the MPM algorithm allows the problem to be mapped efficiently on the GPU for both structured and general graphs.

The following Section provides a brief introduction into the GPU architecture and CUDA programming model. Section 3 discusses the current related work for parallel irregular algorithms as well as maximum flow algorithms. A description of the maximum flow problem as well as the MPM algorithm is provided in Section 4, and Section 5 details our methodology used to map the MPM algorithm on the GPU. Section 6 provides some performance results and lists some hypotheses to explain sources of potential performance degradation in the algorithm. Finally, Section 7 provides our conclusions and briefly discusses some potential future work.

2. CUDA Programming Model
The Compute Unified Device Architecture, or CUDA, provides developers with a framework for general purpose GPU development. This section will briefly cover the exposed GPU architecture as well as the general threading model provided by CUDA. NVIDIA’s CUDA Programming Guide [6] provides further details on any information presented in this section. As we are working with a GPU from the GT200 series, all information included in this section pertains only to that architecture.

2.1. GPU Architecture
The GPU is composed of two major components: (i) The GPU core, and, (ii) The off-chip memory. A series of Streaming Multiprocessors (SMs) compose the GPU core. Each of these SMs contains eight Scalar Processors (SPs). In our work, we use a GTX 260 GPU with 27 SMs and, thus, a total of 216 SPs. Each of the SMs executes following the Single Instruction Multiple Threads (SIMT) model. A modification of the Single Instruction Multiple Data (SIMD) model, SIMT focuses on threads executing identical instructions on different pieces of data.

For data storage, the GPU provides various memories. The largest, and slowest, memory available is the aforementioned off-chip memory, or “global memory”. This memory is read/write accessible to all threads executing across all of the SMs in the GPU. Unfortunately, accessing global memory results in a steep access latency penalty. NVIDIA [6] measures the approximate
latency as 400 to 600 cycles for each access. Further, no automatic caching of data read from
global memory takes place. As a result, this latency hit occurs with each access.

At the SM level, we consider the “shared” memory. Shared memory is as fast as accessing a
register [6] if no bank conflicts exist\(^1\). Clearly, the speed of shared memory results in it being a
tremendous asset for a GPU application. Using shared memory to cache data accessed in global
memory can result in significant performance improvements for a GPU algorithm.

### 2.2. Threading Model

Threads in CUDA are organized into a series of hierarchical groups. At the lowest level is the
thread warp. A thread warp is composed of 32 threads executing on the same SM. Each thread
in a warp must be executing the exact same instruction as every other thread in the same warp.
Threads whose execution paths have diverged are marked as inactive during instructions that
do not pertain to their path. This is, in effect, the SIMT model as discussed in Section 2.1. As a
result, the performance impact of branch-heavy code should become apparent. The worst case,
for example, involves each thread in a warp taking a unique execution path through a block
of code. In this example, the warp’s threads will end up executing the block’s instructions in
serial, rather than in parallel.

One level higher, all warps executing on the same SM are organized into a thread block. Each
thread within a thread block is provided an ID unique only among other threads in the same
thread block. Further, each thread block itself is provided an ID that uniquely identifies it from
all other thread blocks. At the highest level exists the thread grid. The thread grid is composed
of all thread blocks participating in the execution of the kernel.

Shared memory is exclusively allocated to each thread block executing within an SM. For
example, if there are four thread blocks scheduled on a single SM, each thread block can only use
1/4 of the total available shared memory for itself\(^2\). This allocated shared memory is read/write
accessible only to threads within the same thread blocks. Other thread blocks cannot access
shared memory data allocated by another thread block within the same SM.

### 3. Related Work

There exists a number of sequential algorithms that solve the maximum flow problem. Ford
and Fulkerson [7] described one of the first maximum flow algorithms that use the concept of a
flow augmenting path. This algorithm was later improved on by Edmonds and Karp [8]. Later,
Dinic [9] improved on these concepts further and introduced the idea of a layered network.

Malhotra, Pramodh Kumar and Maheshwari’s [4] “MPM algorithm” provides an efficient,
parallelizable way of finding maximum flows within a layered network. There are several phases
to this algorithm which will be explained in Section 4. We consider the MPM algorithm as the
algorithm is very iterative and synchronous — concepts that we believe allow it to map well to
the GPU architecture.

Finally, Goldberg and Tarjan [10, 11] moved away from the layered network concept and
proposed a “push-relabel” algorithm that solves the maximum flow problem. Unlike the previous
algorithms mentioned above, the push-relabel algorithm is very asynchronous in nature. As a
result, we believe it may not be the best-suited maximum flow algorithm for the GPU.

A large amount of research in high performance computing has targeted the GPU in recent
years. Most of the literature, however, has focused on accelerating very regular algorithms on
the GPU. Typically, these regular algorithms display tremendous speedup on the GPU when
compared to sequential CPU code as well as alternative parallel architectures. For example,

---

\(^1\) Shared memory is split into 16 32-bit wide banks, multiple requests for data from the same bank arriving at
the same time are serialized.

\(^2\) All thread blocks allocated for a task are automatically initialized to use the same resources as every other
thread block. Thus, each thread block must use an equal portion of shared memory.
Fujimoto [12] provides an implementation of fast, efficient dense matrix-vector multiplication on the GPU. Fujimoto’s results show a speedup of approximately 32 on the GPU compared to a sequential CPU implementation.

Unfortunately, the current breadth of literature for irregular algorithms on the GPU remains limited when compared to regular algorithms. The literature does, however, contain some existing work on maximum flow algorithms for the GPU. The authors of these papers focus on implementations of Goldberg and Tarjan’s [5] push-relabel algorithm, as it is the maximum flow algorithm with the best-known sequential time complexity.

Hussein et al. [2] first showed a GPU implementation of Goldberg and Tarjan’s [11] push-relabel algorithm. The authors focused on grid graphs, however, which reduces the potential of their algorithm. As a result, Hussein et al. had the ability to develop optimizations that specifically target the structured nature of grid graphs. These optimizations included the lockstep breadth-first traversal.

Following the work of Hussein et al. [2], Vineet and Narayanan [13] detailed their own implementation of the push-relabel algorithm on the GPU. The authors, too, looked only at grid graphs for their implementation. One optimization they investigated involved a method to skip inactive vertices from being processed in a GPU kernel. Vineet and Narayanan also investigated how packing data together impacted the performance of their GPU algorithm. Surprisingly, they showed that their data compaction schemes generally resulted in worse performance than simply leaving the data as-is.

Harish et al. [14] provided a more recent example of a push-relabel GPU algorithm. Unlike Hussein et al. [2] and Vineet and Narayanan [13], Harish et al. [14] considered random graphs. The authors implemented a scan operation that aided in the identification of only those vertices that will be active in the next iteration. Their algorithm then only assigns threads to cover those vertices that are marked as active. Harish et al. further investigate the performance difference between high-degree and low-degree graphs. They concluded that high-degree graphs typically show improved performance over a sequential implementation.

Much of the motivation for our work with the MPM algorithm stems from Kulkarni et al.’s [15] Galois Project. The authors’ designed the Galois Project to look at automatic parallelization for irregular algorithms. Kulkarni et al. [16] show that a mapping towards iterative, synchronous algorithms is ideal for parallel irregular algorithms. The MPM algorithm itself is a very iterative, very synchronous algorithm. Hence, this provides an opportunity to observe how an algorithm with these properties maps and performs on the GPU architecture.

4. A Maximum Flow Algorithm for the GPU

The maximum flow problem begins with a graph, \( G \), containing a set of vertices, \( V \), connected by directional edges, \( E \), with no self-loops. Each edge, \( e \), maintains two values:

(i) \( c(e) \) – The capacity for flow. That is, how many units of flow this edge can support.

(ii) \( f(e) \) – The current amount of flow through this edge.

When \( f(e) = c(e) \) the edge \( e \) is said to be saturated, as no further flow can be sent across this edge. Clearly, the amount of flow reaching a vertex is bound by the amount of flow that can enter, or leave that same vertex across the incoming or outgoing edges. Finally, we define the residual capacity of an edge to be \( \bar{c}(e) = c(e) - f(e) \), which describes how much remaining flow edge \( e \) can still support.

There exists two uniquely identified vertices in \( G \). One vertex, labelled as the source, contains only outgoing edges. As the name implies, it is the source of flow in the graph. Another vertex in the graph, the sink, contains only incoming edges. In the maximum flow problem, we want to find the maximum amount of flow that can be pushed from the source to the sink. A correct solution results in an outflow from the source that is greater than or at least equal to any other
possible solution. Further, a correct solution must satisfy the following rule: the inflow at the sink must be equal to the outflow from the source. That is, every unit of flow leaving the source must, eventually, reach the sink by flowing across a series of valid edge choices.

The MPM algorithm accepts a graph $G = (V, E, s, t)$, where $V$ is a set of vertices, $E$ is a set of weighted edges (the weight represents $c(e)$ for each edge), $s$ is the source vertex and $t$ is the sink vertex.

A layered network structure acts as the base concept behind the MPM algorithm. The layered network, $G_L$, is constructed from $G$ in a manner similar to a breadth-first traversal of a graph. Starting at $s$, we traverse all edges where $\tilde{c}(e) > 0$ (that is, edges that can still support at least one unit of flow) and assign the adjacent vertices a distance label of 1. We then repeat the process for all of the vertices with a distance label of 1, marking the next set of adjacent vertices with a distance label of 2. This process continues until $t$ has been assigned a distance label. Once assigned a distance label, the label is not changed, even if the same vertex is “hit” again during a later edge traversal. Vertices and edges that have been traversed during this breadth-first traversal of $G$ are added to $G_L$.

Once $t$ has been assigned a distance label, any vertices with no outgoing edges must be pruned from $G_L$ (excluding $t$ itself). By pruning a vertex we remove the vertex itself from $G_L$ as well as any edges connecting to this vertex. This pruning effect may then propagate to other vertices as a result of the removal of other edges/vertices. This pruning step ensures that the layered network contains a series of shortest paths: any edge chosen at any layer in $G_L$ is guaranteed to form one step in a shortest path between $s$ and $t$.

Following the pruning step, we must then choose the vertex with the minimum potential, $v_m$, in $G_L$. The potential of a vertex is defined as the minimum of the in or out-potential, where the in-potential is the sum of all $\tilde{c}(e)$ values for each edge moving into this vertex, and the out-potential is the sum of all $\tilde{c}(e)$ values for each edge moving out of this vertex. The potential of $v_m$ is then pushed as flow to $t$ and pulled as flow from $s$. What this step involves is simply pushing this flow across arbitrary forward edges from $v_m$ to the next layer, and across arbitrary backward edges from $v_m$ to the previous layer in $G_L$. This process continues until the flow on either side of $v_m$ has reached $s$ and $t$.

Once flow has reached $s$ and $t$, the algorithm prunes $G_L$ once again. Clearly, at least one vertex, $v_m$, must be removed as flow equal to the potential of $v_m$ was sent through $G_L$ (and, thus, all of the outgoing and/or incoming edges will be saturated). Following the pruning, the algorithm searches for the next $v_m$ and continues sending flow throughout $G_L$.

The process of pushing/pulling flow in $G_L$ terminates when a path from $s$ to $t$ can no longer be found in $G_L$. When this event occurs, the $f(e)$ values for edges in $G$ are updated with the new, updated values from $G_L$. A new layered network is constructed and the algorithm restarts using the updated data for $G$. The algorithm itself terminates when a layered network can no longer be constructed that contains both $s$ and $t$. At this point, the algorithm returns, and a maximum flow possible for $G$ is reported.

4.1. Example

We provide a brief example of a full iteration of the MPM algorithm. Figure 1 provides each step of the algorithm. (a) shows the original graph, $G$. The capacity of each edge is listed above each edge (on the left-hand side for edge $(4, t)$), and the current flow being sent through an edge is listed below each edge. (b) shows the construction of the layered network, $G_L$. In order to start pushing and pulling flow, as shown in (c), the vertex of minimum potential in $G_L$ is first determined. Clearly, vertex 2 has a minimum potential of 1. Hence, 1 unit of flow must be pushed from and pulled to vertex 2. Following this pushing and pulling of flow, edge $(2, t)$ becomes saturated. As a result, it is pruned from $G_L$. Now, vertex 2 no longer has any outgoing edges and, thus, its potential is 0. It, too, is removed from $G_L$ along with edge $(s, 2)$.  

5
In (d), the next vertex with minimum potential is chosen. The vertex chosen is vertex 3, with a potential of 3 (Note that vertex \( t \) was also a valid choice in this scenario, as it, too, has a potential of 3). 3 units of flow are pushed and pulled from vertex 3 to the sink and source. Following this operation, edge \((3, t)\) becomes saturated. Similar to the example in the previous paragraph, this edge is removed and the pruning effect cascades to vertex 3 and edge \((s, 3)\). At this point in time, a path no longer exists between \( s \) and \( t \) in \( G_L \). Thus, this iteration is complete and the amount of flow moved along each edge in \( G_L \) is merged with the current state of \( G \). The algorithm then “restarts”, using the current state of \( G \) as the input data.

5. Solution Methodology
We implement the MPM algorithm on an NVIDIA GTX 260 graphics card using CUDA. As a result of the massive parallelism possible on the GPU, we assign a single vertex in the graph to each thread in the GPU. This allows us to exploit very fine-grained parallelism but still allows our implementation to work on very large graph sizes (in terms of vertex count, \( |V| \)). To mimic the synchronicity of the original sequential algorithm, we decompose the algorithm into a series of sub-algorithms or phases. These phases are as follows:

(i) layered network construction (a modified breadth-first traversal of \( G \)),
(ii) pruning of the layered network,
(iii) searching for the vertex with the minimum potential, and,
(iv) pushing and pulling of flow.

Our implementation divides each phase into a separate GPU kernel. This separation of phases into kernels allows for explicit synchronization between all threads at each step in the algorithm\(^3\).

We focus on the mapping and implementation of the layered network construction phase, an extension to our previous work in [17] where we investigate and analyze the performance of a breadth-first search GPU algorithm. The layered network construction phase for the MPM algorithm is, essentially, a modified breadth-first search. The breadth-first search must be modified, as the layered network construction phase must take into account the \( \tilde{c}(e) \) value of an edge while performing the traversal. This requires a greater degree of computations and memory accesses in order to accomplish.

\(^3\) CUDA does not allow for synchronization between all threads during the execution of a kernel, only threads within a thread block.
Figure 2. Adjacency-list representation of a graph using two arrays.

Rather than create a layered network, $G_L$, which would incur an added degree of performance degradation due to memory accesses, we use the GPU copy of $G$ to manage all aspects of the MPM algorithm. We maintain lightweight state arrays that represent the active vertices and edges in $G_L$ and allow the algorithm to work on the local copy of $G$ directly. The layered network construction phase marks each edge as active in the edge state array during the traversal of the graph. Each traversed vertex in the vertex state array is marked with the distance label of that vertex. This method has the added benefit of not requiring an update to $G$ after each pushing/pulling of flow phase, as $G$’s data is being modified directly.

Furthermore, each edge, rather than storing both the $c(e)$ and $f(e)$ values, simply maintains a record of the $\tilde{c}(e)$ value. This serves to reduce both the memory accesses and computations required when confirming an edge is still useful (as both $c(e)$ and $f(e)$ would be required otherwise). As we still need the $f(e)$ values for the final solution (in order to determine the total amount of flow across each edge), we transfer the $\tilde{c}(e)$ values back to the CPU when the algorithm has completed. The CPU maintains an original copy of $G$ with the original $c(e)$ values. From this, we can compute the final $f(e)$ values for each edge.

In order to improve the overall memory performance of our algorithm, we use a dual-array adjacency list to store $G$. Figure 2 shows the two arrays used. Harish et al. [18, 14] describe this technique as being memory efficient on the GPU. We load vertex-edge offset start values into the shared memory assigned for each thread block. Thread $i$ (covering some vertex $j$) in each thread block then reads the end offset value (that is, the start offset value for the next vertex, $j + 1$) from the shared memory data that is written by thread $i + 1$. The total number of redundant global memory reads required to read in the start and end offsets for each vertex’s edge sets is reduced with this method.

As a brief example, consider the third element in the “vertex-edge offset” array in Figure 2. The value of 3 tells us that the edges for the third vertex begin at element 3 (assuming a zero-indexed array) in the edge destination array. By looking at the next element in the vertex-edge offset array, which is equal to 4, we know that elements up to (but not including) 4 in the edge destination array also define edges for vertex 3. Hence, vertex 4 has a single edge: $e = (3, 5)$.

Algorithm 1 shows the CPU loop used to drive the MPM algorithm. Each of the phases of the MPM algorithm run on the GPU, with the CPU managing the kernel invocations and termination condition checking. As a result, we require only 4 – 8 bytes of data transferred from the GPU to the CPU during each iteration of a phase in order to check termination conditions. Initially, the graph is constructed by the CPU, and then transferred to the GPU. Large-scale transfers back to the CPU only take place when the MPM algorithm has terminated.

The end result is a very synchronous, very iterative algorithm. Synchronization exists between kernel invocations (essentially iterations and phases), and each phase as well as the algorithm itself is entirely iterative in nature.
Algorithm 1 MPM Algorithm Framework

Require: \( G = (V, E, s, t) \)

while A path exists from \( s \) to \( t \) in \( G \) do

\( G_L \leftarrow G \) partitioned into a layered network

while A path exists from \( s \) to \( t \) in \( G_L \) do

Find vertex \( v \) with min. vertex potential

Push/pull flow from \( v \) to \( s, t \)

Prune \( G_L \)

end while

end while

6. Results

All of the results provided in this Section were gathered from the parallel GPU algorithm running on a GTX 260, and a sequential CPU algorithm running on an Intel Core 2 Duo at 3.0Ghz. The sequential code was written to be optimized for sequential execution but was not optimized specifically for the CPU architecture.

We provide Figure 3 as an example of the execution time for one phase of the algorithm — the construction of the layered network. We test the execution time on random graphs with an average vertex degree of 5. The measurement includes only the time required to construct a layered network on the GPU. It is clear that the GPU outperforms the CPU by a significant margin for larger graphs. The GPU algorithm displays a speedup of approximately 10 for the largest graph size tested.

![Layer Network Construction Execution Times](image)

Figure 3. Layered Network Construction Execution Times

We further test the MPM algorithm as a whole on graphs with a vertex degree of 10. The results, provided in Figure 4, show that the GPU implementation provides a greater degree of performance than the sequential CPU implementation. The speedup for the GPU MPM algorithm as a whole is, on average, 8, when compared to the sequential CPU version.

An immediate issue that is likely reducing performance is the nature of the memory accesses. Phases (1), (2) and (4) (as listed in Section 5) require a significant level of unpredictable memory accesses during the algorithm in order to update important data related to the state of each vertex and edge. As described by Ruetsch and Micikevicius [1], the performance of a GPU algorithm can be significantly impacted by the performance of memory accesses. As a result, the high amount of unpredictable memory accesses will likely be the root cause of potential performance degradation in the algorithm. Unfortunately, due to the irregularity of these phases, the act of improving memory performance becomes tremendously difficult if not impossible.
Another issue is the load balancing between thread warps. Currently, all threads covering all vertices are launched with each kernel invocation, regardless of whether or not every thread will be active in the coming iteration. This results in a load imbalance amongst threads in a thread warp. Perhaps not all of the vertices covered by threads in a warp will be active in the next iteration, and, due to this, less parallelism is being exploited. Consider an example where only 20 out of 32 threads in a warp have active vertices. In this scenario, only 20 out of the 32 threads will be executing. 12 threads will be idling, wasting resources that could be put to use doing useful work. Of course, the act of preparing and maintaining a list of vertices that will be active in the next iteration may result in an equal level of performance degradation in and of itself. As discussed in the next Section, we leave experimentation of this idea as future work.

7. Conclusions and Future Work
We have presented the details of a GPU implementation of the MPM algorithm. Our results show that the GPU algorithm can achieve, on average, approximately 8 times speedup over a sequential CPU implementation. Furthermore, one phase of the algorithm, the construction of a layered network, shows a speedup of 10 when compared to a sequential CPU implementation. Despite the architectural inefficiencies and difficulties associated with parallel irregular algorithms on the GPU, the GPU MPM algorithm can still achieve greater performance than a CPU algorithm.

We believe there still exists some future work to be performed on our implementation in order to improve the performance further. One such modification that may result in improved performance is the reduction in threads that are not performing useful work. This will serve to reduce branch divergence as well as ensure that each thread within each SM is performing useful work, rather than sitting idle. The experimentation of this hypothesis is left as future work.

References
[1] Ruetsch G and Micikevicius P 2009 Optimizing matrix transpose in CUDA Published in the CUDA SDK 2.3 package
[2] Hussein M, Varshney A and Davis L 2007 On implementing graph cuts on CUDA First Workshop on General Purpose Processing on Graphics Processing Units (Boston, MA, USA)
[3] Fleischer L 1998 Faster algorithms for the quickest transshipment problem with zero transit times Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms (SODA 98) (San Francisco, California, United States: Society for Industrial and Applied Mathematics) pp 147–156
[4] Malhotra V M, Kumar M P and Maheshwari S N 1978 Information Processing Letters 7 277–278
[5] Goldberg A V and Tarjan R E 1988 Journal of the ACM 35 921–940
[6] NVIDIA 2010 Cuda 3.0 Programming Guide NVIDIA
[7] Ford L and Fulkerson D 1956 *Canadian Journal of Mathematics* **8** 399–404
[8] Edmonds J and Karp R 1972 *Journal of the Association for Computing Machinery* **19** 248–264
[9] Dinic E A 1980 *Soviet Math. Dokl.* **11** 1277–1280
[10] Goldberg A 1987 *Efficient Graph Algorithms for Sequential and Parallel Computers* Ph.D. thesis Laboratory of Computer Science, MIT
[11] Goldberg A and Tarjan R 1988 *Journal of the Association for Computing Machinery* **35** 921–940
[12] Fujimoto N 2008 Faster matrix-vector multiplication on GeForce 8800GTX *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)* (Miami, Florida, USA: IEEE) pp 1–8
[13] Vineet V and Narayanan P J 2008 CUDA cuts: Fast graph cuts on the GPU *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (Anchorage, AK: IEEE) pp 1–8
[14] Harish P, Vineet V and Narayanan P 2009 Large graph algorithms for massively multithreaded architectures Technical Report IIT/TR/2009/74 International Institute of Information Technology Hyderabad Hyderabad, India
[15] Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K and Chew L P 2007 Optimistic parallelism requires abstractions *Proceedings of Programming Language Design and Implementation 2007* (San Diego, CA, USA: ACM) pp 211–222
[16] Kulkarni M, Burtscher M, Inkulu R, Pingali K and Cascaval C 2009 How much parallelism is there in irregular applications? *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, NC, USA: ACM) pp 3–14
[17] Solomon S and Thulasiraman P 2010 Performance study of mapping irregular computations on GPUs *The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10)* (Atlanta, USA: IEEE)
[18] Harish P and Narayanan P 2007 Accelerating large graph algorithms on the GPU using CUDA *14th International Conference on High Performance Computing (HiPC 2007)* (Goa, India: Springer-Verlag) pp 197–208