CHAPTER

5 Efficient CUDA Algorithms for the Maximum Network Flow Problem

Jiadong Wu, Zhengyu He, and Bo Hong

In this chapter, we present graphical processing unit (GPU) algorithms for the maximum network flow problem. Maximum network flow is a fundamental graph theory problem with applications in many areas. Compared with data-parallel problems that have been deployed onto GPUs, the maximum network flow problem is more challenging for GPUs owing to intensive data and control dependencies. Two GPU-based maximum flow algorithms will be presented in this chapter — the first one is asynchronous and lock free, whereas the second one is synchronized through the precoloring technique. We will demonstrate in this chapter that, with careful considerations in algorithm design and implementation, GPUs are also capable of accelerating intrinsically data-dependent problems.

5.1 INTRODUCTION, PROBLEM STATEMENT, AND CONTEXT

The maximum network flow problem is a fundamental graph theory problem. Given a directed graph with two distinct nodes, source and sink, and the capacity constraints on each edge, the problem aims to maximize the amount of flow that can be sent from the source to the sink. The maximum flow problem has many applications in different areas. For example, as a fundamental graph problem, it can be used to solve other graph problems such as disjoint paths, bipartite matching, and circulation with demand. It can also be used in some industrial applications like segmentation in image processing, routing in very large scale integration (VLSI) design and scheduling in air transportation.

Historically, multiple algorithms exist for the problem. Early solutions to the maximum flow problem are based on the augmenting path method due to Ford and Fulkerson [1], which was pseudo-polynomial and was later improved by carefully choosing the order in which augmenting paths are selected (e.g., the $O(|V||E|^2)$ algorithm by Edmonds and Karp [2] and the $O(|V|^2|E|)$ algorithm by Dinitz [3]). The concept of preflow was introduced by Karzanov in [4], which leads to an $O(|V|^3)$ algorithm. Goldberg et al. designed the push-relabel algorithm [5] with $O(|V|^2|E|)$ operations and further improved the complexity bound by using various techniques. Among those solutions, Goldberg's push-relabel algorithm is relatively easier to parallelize than other augmenting path-based algorithms, thus a few attempts have been made to parallelize it such as Anderson et al. [6] and Bader et al. [7]. Although these algorithms have demonstrated good execution speed on multicore processors, they share the common feature of using locks to protect every push and relabel operation in its entirety, which makes the algorithms unsuitable for the CUDA programming model.

GPU Computing Gems. DOI: 10.1016/B978-0-12-385963-1.00005-8

The simplicity of CUDA's programming model, in fact, projects a number of challenges to design maximum flow algorithms for the multicore CUDA platform. First, CUDA does not natively provide a critical code section construct, which is essential to the correctness of the existing parallel maximum flow algorithms. It is possible to implement locks and critical sections in CUDA by explicitly using atomic memory operations. However, a study is needed to investigate the performance of such an approach. Second, multiprocessors in CUDA work in an Single Instruction Multiple Data (SIMD) fashion, where best performance is achieved when a warp of threads always takes the same execution path. Divergence among cores in the same multiprocessor results in serialization of the different paths taken and hence can cause performance penalties. The SIMD execution model becomes a significant challenge for any inherently divergent task, including the maximum flow problem.

To efficiently solve the maximum flow problem on CUDA, the algorithm should avoid using locks. In this chapter, we present two novel lock-free variations of the parallel push-relabel algorithm. The first algorithm solves the maximum flow problem by using atomic operations to perform the push and relabel operations asynchronously. The second algorithm works on precolored graphs and avoids race condition through barriers. Experiments using the NVIDIA C1060 GPU show that, despite the intrinsic challenges of data dependencies and divergent execution paths, both algorithms are able to achieve at least 3 times, and up to 8 times, speed-ups over implementations on a quad-core Intel Xeon CPU.

In this chapter we demonstrate that, with carefully designed algorithms and implementation techniques, we can use CUDA to accelerate complicated applications.

5.2 CORE METHOD

Maximum flow algorithms based on the push-relabel technique typically consist of two stages. The first stage searches for a minimum cut and the value of a maximum flow. The second stage constructs a valid maximum flow by returning possible excessive flows back to the source (the vertices may have excessive flow upon completion of the first stage). The complexity of the first stage is $O(|V|^3)$ or $O(|V|^2|E|)$ depending on the vertex processing order. The second stage uses $O(|E|log|V|)$ operations, which takes much less time than the first stage. For applications searching for the minimum cut such as some computer vision applications [8], the second stage is not needed at all. Because the first stage dominates the execution time, the study in this chapter will focus on the first stage. Before presenting the CUDA algorithms for solving the max-flow problem, we first present the problem formulation and briefly review the sequential push-relabel algorithm.

Given a direct graph $G(V,E)$ with source $s \in V$ and sink $t \in V$, in which every edge has capacity c_{uv}, the function f is called a flow if it satisfies $f(u,v) \leq c_{uv}, f(u,v) = -f(v,u)$ for $u,v \in V$. The residual capacity $c_f(u,v)$ is given by $c_{uv} - f(u,v)$, and the residual graph of G induced by f is $G_f(V,E_f)$ for $E_f = \{(u,v)|u,v \in V, c_f(u,v) > 0\}$: the same vertex set but only edges with residual capacity. Example of flows on a simple graph and its residual graph is given in Figure 5.1. The maximum flow problem is to search for function f which maximizes the value of $|f| = \sum_{v \in V} f(s,v)$.

The sequential push-relabel algorithm, due to Goldberg [5], moves flows from the source to the sink using localized operations. The source first *pushes* as much flow as possible to its neighbor vertices causing those vertices to temporarily have more incoming flows than outgoing flows. In the algorithm, a vertex v is called *active* when it has excessive incoming flows (i.e., $\sum_{u \in V} f(u,v) > 0$). The active

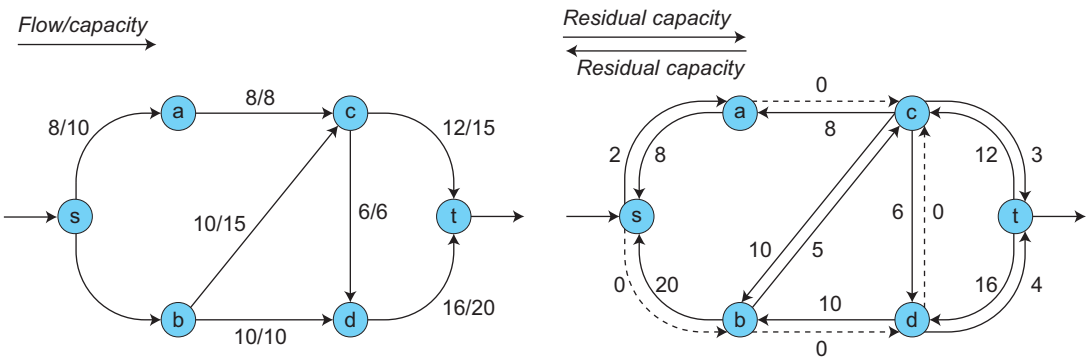

FIGURE 5.1

A flow f in graph G and the residual graph G_f. Dotted edges have 0 residual capacity and are therefore not part of the residual graph.

vertices will be selected one at a time, in certain order, to push the excessive flows to their neighbors, thus *activating* more vertices. In this algorithm, each vertex is assigned an integer valued *label*, which is an estimate of the vertex's distance to the sink. The labels of the source and the sink are fixed at $|V|$ and 0, respectively. All the other nodes have an initial label of 0. Vertex labels are used to guide the push operations to move the flows towards the sink. A push operation always pushes to a neighbor with a lower label. If an active vertex cannot push any flow because it has a lower label than all its neighbors in the residual graph, the vertex will be *relabeled* to be one plus the minimum of all its neighbors. The algorithm repeatedly applies the push and relabel operations until there are no active vertices. When the algorithm terminates, flows will either be delivered to the sink or returned to the source, and the flows at the sink is the maximum flow of the graph.

The original push-relabel algorithm is a sequential algorithm designed for single-threaded execution. A straight-forward parallelization of this algorithm is to assign one thread for each node and let the threads do push-relabel simultaneously, but there will be race conditions if push and relabel operations are simultaneously applied to the same vertex. When such race condition occurs, two push operations may simultaneously and incorrectly modify the excessive flow of a single vertex, or a push operation may send flow to a vertex that is simultaneously being relabeled, thus violating the requirement that flow is always pushed to neighbors with a lower label.

To efficiently implement CUDA max-flow algorithms, the race conditions need to be handled without using locks. In a recent study, we proposed an asynchronous lock-free push-relabel algorithm targeting multicore processors, which will serve as a good starting point for a CUDA port. The algorithm differs from the original push-relabel algorithm in the following three aspects: (1) the push operation sends flow to the lowest neighbor rather than to an arbitrary lower neighbor; (2) each push and relabel operation is redesigned with multiple atomic instructions, thus eliminating the needs of locking the vertices; and (3) the termination condition examines the value of $e(s) + e(t)$ instead of the existence of overflowing vertices. These modifications allow the algorithm to be executed asynchronously by multiple threads, while the complexity $O(|V|^2|E|)$ stays the same as that of the original push-relabel algorithm. Detailed proof of correctness and complexity can be found in [9]. Our first

CUDA max-flow algorithm is a variation of this asynchronous algorithm, with updates to explore CUDA parallelism.

We also construct a precoloring-based lock-free variation of the push-relabel algorithm, which can solve maximum flow problems on spares graphs with higher efficiency than the asynchronous algorithm. In the new algorithm, before the push-relabel starts, the edges are precolored so that any two edges will have different colors if they share a common vertex. In each iteration of the push relabel, only edges of the same color are selected as candidates for the push operations. Because edges with the same color do not share any vertices, data races will not occur and the original push operation can be used; thus, regular add and subtract instructions can be used. The algorithm will loop through all the colors and execute a global thread barrier between two colors.

In our asynchronous algorithm as well as the precoloring-based algorithm, the progress at one thread does not need to synchronize with any other threads. While thread u is executing the push-relabel code for vertex u, all the other threads are executing the same code for their own vertices. Such properties expose maximum parallelism in the execution of the algorithm and makes the algorithm suitable for the CUDA programming model.

5.3 ALGORITHMS, IMPLEMENTATIONS, AND EVALUATIONS

The CUDA-based parallel push-relabel algorithm is presented in Algorithm 1, which has two important functions: push relabel and global relabel. Push relabel runs on CUDA. Global relabel is the heuristic running periodically on CPU, which improves the overall effectiveness of push and relabeling. We discuss the details of global relabel later in this section.

The initialization stage of the algorithm is to prepare c_f, h, and e based on the target graph. The algorithm will then enter the main loop (lines 3–8 in Algorithm 1), where the required data are first transferred to the GPU, and the CUDA kernel is then launched to concurrently execute the push and relabel operations. After the kernel executes a certain number of cycles, c_f, h, and e are transferred back to the CPU's main memory. The CPU will perform the global relabeling by calling global–relabel–CPU. The global variable *ExcessTotal* is used to track the total amount of excessive flow in the residual graph. The algorithm terminates when $e(s) + e(t)$ becomes equal to *ExcessTotal*, which is equivalent to the condition that no active vertices exist in the graph (except for the source and sink). The loop will be iterated repeatedly until the termination condition is satisfied. When the algorithm terminates, $e(t)$ stores the value of the maximum flow.

Algorithm 1: Implementation of the parallel push-relabel algorithm using CUDA

```
1: Initialize e, h, c_f, and ExcessTotal
2: Copy e and c_f from the CPU's main memory to the CUDA device global memory
3: while e(s) + e(t) < ExcessTotal do
4:     copy h from the CPU main memory to the CUDA device global memory
5:     call push-relabel() kernel on CUDA device
6:     copy c_f, h, and e from CUDA device global memory to the CPU's main memory
7:     call global-relabel() on CPU
8: end while
```

The asynchronous push-relabel is presented in Algorithm 2. It uses read-modify-write atomic instructions that are supported by CUDA. The kernel launches one thread for every vertex (except for the source and the sink) and concurrently executes the push and relabel operations. Every thread will continuously perform push or relabel operations until the vertex u becomes inactive: $e(u) = 0$ or $h(u) > n$.

Algorithm 2 augments the original push-relabel algorithm by pushing to the lowest neighbor (whereas in the original algorithm, it pushes to any lower neighbor). It can be shown that even though different threads may execute their push and relabel operations in an arbitrary order, the kernel in Algorithm 2 eliminates the impact of data races and finds the maximum flow with $O(|V|^2|E|)$ operations.

Although the asynchronous lock-free algorithm is compatible with CUDA, the parallelization efficiency is affected by three factors: (1) the number of available push and relabel operations, (2) the overheads of data transfer between the CPU and the GPU, and (3) the efficiency of atomic operations on CUDA. In fact, the number of active vertices is often small for sparse graphs. Experiment results show that, for a Genrmf graph (see Section 5.4 for a description of the graph) with 262,144 vertices and 1,276,928 edges, although 262,144 threads can be launched simultaneously, there are at most 1,400 active vertices at a time. Most of the time, especially when the algorithm initially starts, only hundreds

Algorithm 2: Implementation of asynchornous push-relabel kernel on CUDA device

```
1:  cycle = KERNEL_CYCLES
2:  while cycle > 0 do
3:      if e(u) > 0 and h(u) < n then
4:          e' = e(u)
5:          h' = ∞
6:          for all (u,v) ∈ E_f do
7:              h'' = h(v)
8:              if h'' < h' then
9:                  v' = v
10:                 h' = h''
11:             end if
12:         end for
13:         if h(u) > h' then
14:             d = min(e', c_f(u,v))
15:             AtomicAdd(c_f(v',u), d)
16:             AtomicSub(c_f(u,v'), d)
17:             AtomicAdd(e(v'), d)
18:             AtomicSub(e(u), d)
19:         else
20:             h(u) = h' + 1
21:         end if
22:     end if
23:     cycle = cycle − 1
24: end while
```

of vertices are active. Thus, many CUDA threads will be idle during the computation. In addition, Algorithm 2 needs to execute AtomicAdd and AtomicSub through the global memory, which is slower than the regular add and subtraction instructions executing inside the stream processors.

To overcome these limitations, another algorithm for push relabel is presented in Algorithm 3. Before the kernel starts, the edges are precolored so that any two edges will have different colors if they share a common vertex. In each iteration of the push-relabel kernel, only edges of the same color are selected as candidates for the push operations. Because edges with the same color do not share any vertices, data races will not occur, and the original push operation can be used; thus, regular add and

Algorithm 3: Implementation of precoloring-based synchronized push-relabel kernel

```
1:  cycle = KERNEL_CYCLES
2:  while cycle > 0 do
3:      if e(u) > 0 and h(u) < n then
4:          lflag = 1
5:          e' = e(u)
6:          h' = ∞
7:          for all (u,v) ∈ E_f do
8:              h'' = h(v)
9:              if h'' < h' then
10:                 lflag = 0
11:                 v' = v
12:                 break
13:             end if
14:         end for
15:     else
16:         lflag = 0
17:     end if
18:     barrier();
19:     for all cr ∈ COLORS do
20:         if color(v') = cr and lflag = 0 then
21:             d = min(e', c_f(u,v))
22:             c_f(v',u) = c_f(v',u) + d
23:             c_f(u,v') = c_f(u,v') − d
24:             e(v') = e(v') + d
25:             e(u) = e(u) − d
26:         end if
27:         barrier();
28:     end for
29:     if lflag = 1 then
30:         h(u) = h(the lowest neighborer of u) + 1
31:     end if
32:     cycle = cycle − 1
33: end while
```

subtract instructions can be used. The algorithm will loop through all the colors and execute a global thread barrier between two colors.

A graph can be colored using a simple greedy heuristic with $O(VE)$ operations. A reasonable amount of time will be saved in the later push-relabel stage. For example, a Genrmf graph with 2,515,456 vertices and an average vertex degree of 5 can be colored in 2 seconds with 11 colors, and the precoloring-based push-relabel algorithm takes about 60 seconds to compute its maximum flow, while the asynchronous push-relabel algorithm takes about 80 seconds to do it.

The efficiency of the global thread barrier (between two colors) needs some special consideration. Currently, CUDA threads within a block can communicate via shared memory or the global memory. In the CUDA programming model, the __syncthreads() function ensures in-block synchronization. However, there is no explicit support for inter-block barriers. Therefore, such a global thread barrier can be implemented only indirectly. CUDA enforces an implicit barrier between kernel launches. During kernel launch, a CUDA thread synchronize function is called implicitly on the CPU, which waits for all the threads in the previous kernel to complete. Alternatively, inter-block barriers can be achieved within the GPU by allowing threads to communicate via the global memory. Both lock-based and lock-free algorithms can be used for such a barrier. In the study by Shucai Xiao et al. [10], in-GPU barriers demonstrate relatively better performance over kernel-launch barriers only if the __threadfence() function is omitted. However, without a __threadfence() function, a thread arriving at the barrier may still have some pending memory operations waiting to be updated to the global memory, which breaks the correctness guarantee of barriers. In addition, according to the experimental results in [10], in-GPU barriers, even without using __threadfence() (which renders the barriers incorrect), will still be outperformed by kernel-launch barriers when the number of blocks exceeds the number of streaming processors. In the precoloring-based maximum flow algorithm, the correctness of the global barrier is critical, and the number of blocks needed is fairly large. As a result, in the implementation of precoloring-based algorithm, the global thread barrier is implemented by iteratively launching the kernels.

Previous studies suggested two heuristics — global relabeling and gap relabeling — to improve the practical performance of the push-relabel algorithm. The height h of a vertex helps the algorithm to identify the direction to push the flow towards the sink or the source. The global relabeling heuristic updates the heights of the vertices with their shortest distance to the sink. This process can be performed by a backward breadth-first search (BFS) from the sink or the source in the residual graph [11]. The gap relabeling heuristic developed by Cherkassky also improves the practical performance of the push-relabel method (though not as effective as global relabeling [11]).

In sequential push-relabel algorithms, the global relabeling heuristic and gap relabeling heuristic are executed by the same single thread that executes the push and relabel operations. Race conditions therefore do not exist. For parallel implementation of the push-relabel algorithms, the global relabeling and gap relabeling have been studied by Anderson [6] and Bader [7], respectively. Both implementations lock the vertices to avoid race conditions. The global or gap relabeling, push, and relabel operations are therefore pairwise mutually exclusive. Unfortunately, the lack of efficient locking primitives on CUDA makes such technology infeasible for the CUDA-based algorithms. Hussein [8] proposed a lockstep BFS to perform parallel global relabeling, but it was shown in [8] that this design was very slow. To overcome these obstacles, global relabeling is performed on the CPU side in this chapter. This is presented in Algorithm 4.

Algorithm 4: Implementation of global-relabel function on CPU

```
1:  for all (u,v) ∈ E do
2:      if h(u) > h(v) + 1 then
3:          e(u) = e(u) − c_f(u,v)
4:          e(v) = e(v) + c_f(u,v)
5:          c_f(v,u) = c_f(v,u) + c_f(u,v)
6:          c_f(u,v) = 0
7:      end if
8:  end for
9:  do a backwards BFS from sink and assign the height function with each vertex's BFS tree level
10: if not all the vertices are relabeled then
11:     for all u ∈ V do
12:         if u is not relabeled or marked then
13:             mark u
14:             ExcessTotal = ExcessTotal − e(u)
15:         end if
16:     end for
17: end if
```

In Algorithm 4, global relabeling is performed by the CPU periodically. The frequency of global relabeling can be adjusted by changing the value of KERNEL_CYCLES. After every thread finishes KERNEL_CYCLES of push or relabel operations, c_f will be transferred from the CUDA global memory to the main memory of CPU along with h and e. If the termination condition is not satisfied, the global-relabel function will assign a new label for each vertex based on this topology of the residual graph (derived from c_f).

5.4 FINAL EVALUATION

To evaluate the performance of the presented maximum flow algorithms, we tested the CUDA algorithms on an NVIDIA Tesla C1060 GPU card and tested the pthread-based CPU algorithm on an Intel Xeon E5520 processor with four threads. The software environment is a Linux kernel version 2.6.18 with CUDA toolkit version 3.1. Both implementations were tested with five typical types of graphs which were also used in the first DIMACS Implementation Challenge [12].

1. Acyclic-dense graphs: These graphs are complete directed acyclic-dense graphs: each vertex is connected to every other vertex. Graphs of 2,000, 4,000, and 6,000 vertices were tested.
2. Washington-RLG-long graphs: These graphs are rectangular grids of vertices with w rows and l columns. Each vertex in a row has three edges connecting to random vertices in the next row. The source and the sink are external to the grid, the source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink. We tested the graphs of $w = 512$, $l = 1,024$ (524,290 vertices and 1,572,352 edges), $w = 768$, $l = 1,280$ (983,042 vertices and 2,948,352 edges) and $w = 1,024$, $l = 1,536$ (1,572,866 vertices and 4,717,568 edges).

3. Washington-RLG-wide graphs: These graphs are the same as Washington-RLG-long graphs except for the values of w and l. Each row in the Washington-RLG-wide graphs is wider. We tested the graphs of $w = 512$, $l = 512$ (262,146 vertices and 785,920 edges), $w = 768$, $l = 768$ (589,826 vertices and 1,768,704 edges) and $w = 1,024$, $l = 1,024$ (1,048,578 vertices and 3,144,704 edges).
4. Genrmf-long graphs: These graphs are comprised of $l1$ square grids of vertices (frames) each having $l2 \times l2$ vertices. The source vertex is at a corner of the first frame, and the sink vertex is at the opposite corner of the last frame. Each vertex is connected to its grid neighbors within the frame and to one vertex randomly chosen from the next frame. We tested the graphs of $l1 = 24$, $l2 = 192$ (110,592 vertices and 533,952 edges), $l1 = 32$, $l2 = 256$ (262,144 vertices and 1,276,928 edges), $l1 = 44$, $l2 = 352$ (681,472 vertices and 3,342,472 edges), $l1 = 56$, $l2 = 448$ (1,404,928 vertices and 6,921,152 edges) and $l1 = 68$, $l2 = 544$ (2,515,456 vertices and 12,424,688 edges).
5. Genrmf-wide graphs: The topology is the same as Genrmf-long graphs except for the values of $l1$ and $l2$. Frames are bigger in Genrmp-wide graphs than in Genrmp-long graphs. We tested the graphs of $l1 = 48$, $l2 = 48$ (110,592 vertices and 541,440 edges), $l1 = 64$, $l2 = 64$ (262,144 vertices and 1,290,240 edges), $l1 = 84$, $l2 = 84$ (292,704 vertices and 2,928,240 edges), $l1 = 108$, $l2 = 108$ (1,259,712 vertices and 6,240,240 edges) and $l1 = 136$, $l2 = 136$ (2,515,456 vertices and 12,484,800 edges).

The experimental results are illustrated in Figures 5.2–5.4. In the figures, cpu-async stands for CPU-based asynchronous algorithm; gpu-async and gpu-color stand for asynchronous CUDA algorithm and precoloring-based CUDA algorithm, respectively.

Figure 5.2 shows the results on acyclic-dense graphs. gpu-async outperforms cpu-async by about three times. Better results of 4 times speed-ups are observed on Washington-RLG graphs as shown in Figure 5.3.

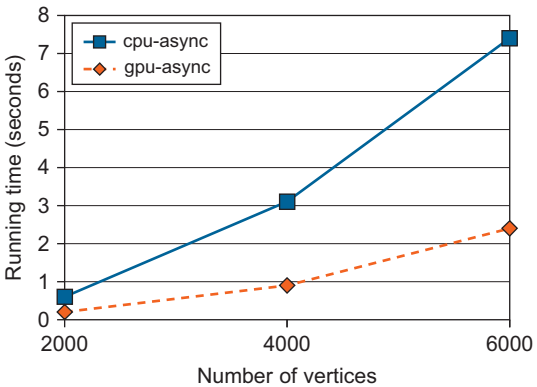

FIGURE 5.2

Experimental results on Acyclic-Dense graphs.

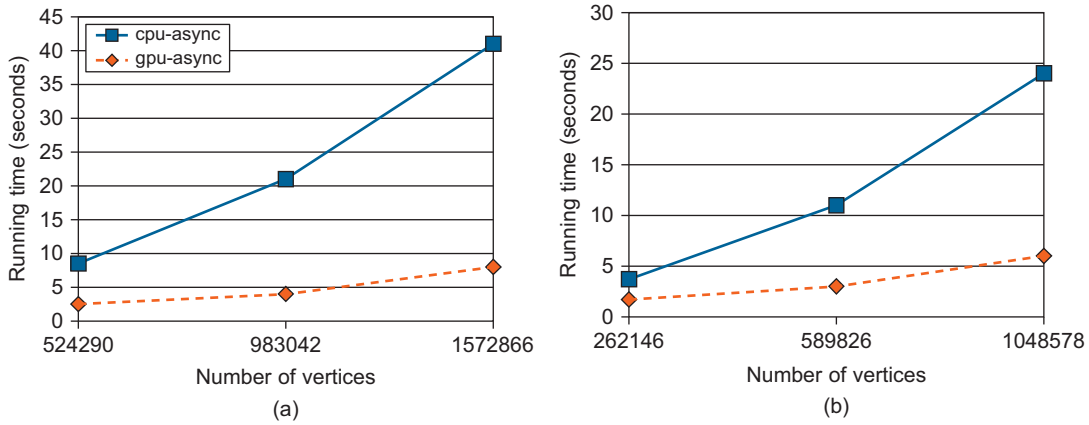

FIGURE 5.3

Experimental results on Washington-RLG graphs.

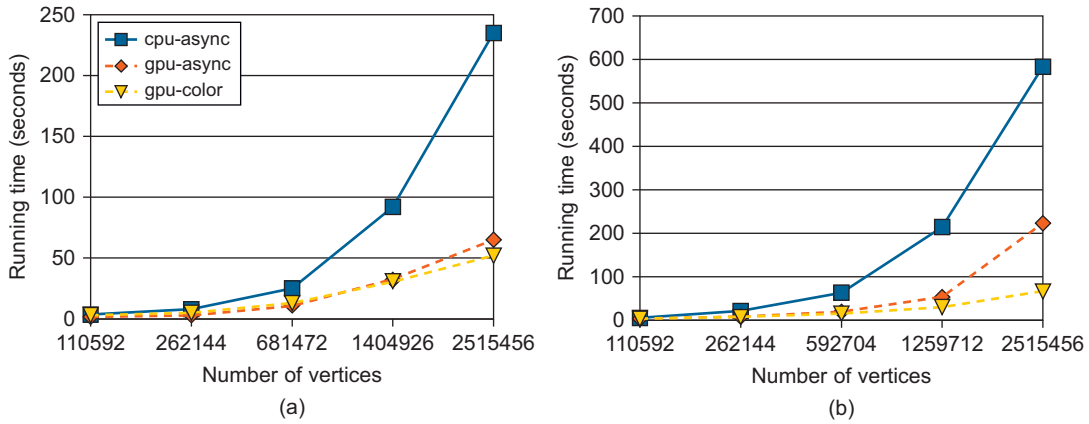

FIGURE 5.4

Experimental results on Genrmf graphs.

Figure 5.4 shows the results on Genrmf graphs. gpu-async and gpu-color still outperform cpu-async by at least three times. Especially on the Genrmf-wide graphs, the gpu-color algorithm achieved a speed-up of more than eight times.

5.5 FUTURE DIRECTIONS

In this chapter, we presented two efficient CUDA algorithms to solve the maximum network flow problem. The asynchronous lock-free algorithm exhibits good performance. And the precoloring-based

algorithm is a reliable alternative, it outperforms the asynchronous algorithm for certain sparse graphs. This chapter presents a good example of the capacity and potential of multicore GPU parallel platforms in accelerating graph theory problems as well as other complex data-dependent applications. The performance can be further improved by incorporating various graph-specific techniques and by carefully allocating the vertex data in the global memory to coalesce memory accesses. Because a vertex may push flow to an arbitrary neighbor in a general graph, which will result in a random memory access pattern, we expect memory coalescing to be difficult for such graphs. However, we expect memory coalescing to work well for highly regular graphs with low-vertex degrees (where the storage of the adjacency list/matrix can be compact and regular).

The precoloring-based algorithm, as expected, does achieve higher efficiency and better scalability over the asynchronous algorithm for some sparse graphs. However, limitation also exists. For any graph, if the maximum degree of vertices is large, then we have to color it with a large numbers of colors. In such case, we will get little parallelism within each one of the colors, which affects the performance of this algorithm adversely. For example, although the Washington RLG graphs have an average vertex degree of 5, the source and sink in these graphs always have extrodinary large degrees. So, we cannot effectively apply the general precoloring-based algorithm on it. A possible way to overcome this limitation is to weigh the colors. Each color has a weight indicating how many vertices it has, and the weight is proportionally connected to the number of push-relabel iterations that will be applied.

References

[1] L.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.

[2] J. Edmonds, R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. ACM 19 (1972) 248–264.

[3] E. Dinic, Algorithm for solution of a problem of maximum flow in networks with power estimation, Sov. Math. Dokl. 11 (1970) 1277–1280.

[4] A.V. Karzanov, Determining the maximal flow in a network by the method of preflows, Sov. Math. Dokl. 15 (1974) 434–437.

[5] A.V. Goldberg, Recent developments in maximum flow algorithms (invited lecture), in: SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory, Springer-Verlag, London, UK, 1998, pp. 1–10.

[6] R.J. Anderson, J.C. Setubal, On the parallel implementation of Goldberg's maximum flow algorithm, in: SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, ACM, New York, 1992, pp. 168–177.

[7] D. Bader, V. Sachdeva, A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic, in: ISCA International Conference on Parallel and Distributed Computing Systems, Phoenix, AZ, 2005, pp. 41–48.

[8] M. Hussein, A. Varshney, L. Davis, On Implementing Graph Cuts on CUDA, in: First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA, 2007.

[9] B. Hong, Z. He, An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic, IEEE Trans. Parallel Distrib. Syst. 22 (6) (2011) 1025–1033, doi: 10.1109/TPDS.2010.156, ISSN: 1045–9219.

[10] S. Xiao, W. Feng, Inter-block GPU communication via fast barrier synchronization, in: IPDPS 09: IEEE International Parallel and Distributed Processing Symposium, Atlanta, GA, 2005, pp. 1–12, doi: 10.1109/IPDPS.2010.5470477, ISSN: 1530–2075.

[11] A.V. Goldberg, Recent developments in maximum flow algorithms (invited lecture), in: SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory, Springer-Verlag, London, UK, 1998, pp. 1–10.

[12] D.S. Johnson, E.C.C. McGeoch (Eds.), Network Flows and Matching: First Dimacs Implementation Challenge, DIMACS Ser. Discrete Math. Theor. Comput. Sci. 12 (1993), American Mathematical Soc., Providence, RI.