Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms

Zhengyu He, Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology
zhengyu.he,bohong@gatech.edu

***Abstract*—The maximum flow problem is a fundamental graph theory problem with many important applications. Max-flow algorithms based on the push-relabel method are known to have better complexity bound and faster practical execution speed than others. However, existing push-relabel algorithms are designed for uniprocessors or parallel processors that support locking primitives, thus making it very difficult to apply the push-relabel technique to CUDA-based GPUs. In this paper, we present a first generic parallel push-relabel algorithm for CUDA devices. We model the parallelization efficiency of the algorithm, which reveals that, for a given input graph, the level of parallelism varies during the execution of the algorithm. To maximize the execution efficiency, we develop a dynamically tuned algorithm that utilizes both CPU and GPU by adaptively switching between the two computing units during run time. We show that algorithm finds the maximum flow with $O(|V|^2|E|)$ operations (summed over both the CPU and the GPU). Extensive experimental results show that the new algorithm is up to 2 times faster than the push-relabel algorithm by Goldberg et al.**

I. Introduction

The maximum flow (max-flow) problem is a fundamental graph theory problem with applications in many areas. Over the years, many algorithms have been developed for this problem with continuously improving complexity bounds. The push-relabel algorithm was originally designed for single-threaded implementations and has the best complexity bound known so far. It is also a good candidate for parallelization due to its localized data access patterns.

Practical implementations of parallel algorithms have also been investigated intensively. Anderson and Setubal [1] augmented the push-relabel algorithm with a *global relabeling* operation. Bader et al. [2] designed a parallel implementation using gap relabeling heuristic with considerations in the cache performance of the push-relabel algorithm. Both implementations have demonstrated good execution speed. These parallel implementations, however, share a few common features that impact the efficiency of the parallelization:

1) Parallel push-relabel algorithms cannot enforce a strict order (e.g. FIFO or the highest-label-first) when processing the vertices. Such orders have been demonstrated to lower the complexity bound of the algorithms.
2) Extra overheads are associated with the load balancing mechanism, lock-based critical section contention, and cache misses due to the invalidation.

As a result, it was observed in [1], [2] that often times the parallel implementations cannot outperform the sequential push-relabel algorithm due to Goldberg [3]. To overcome such limitations and fully exploit the parallelism in the max-flow problem, an alternative method would be to improve the scalability of the algorithm so that we can involve a large number of processors to counteract the impact of parallelization overheads. Towards this objective, we developed a lock-free push-relabel algorithm in [4] that aims at releasing the processors from critical section contention and assuring the independence of the computation tasks. The algorithm allows more processors to be involved in the computation. However, practical multi-core systems are often equipped with a limited number of processor cores (2 or 4 typically, and 6 on some recent high-end systems), and cannot support a sufficient number of threads to fully exploit the potential of the algorithm.

GPU has recently become an essential high performance computing device. It has attracted intensive research interests since it is equipped with a large number of processor cores (though each core is not as powerful as a CPU core). For applications with many independent computation tasks and thus suitable for this architecture, GPU computing has demonstrated substantial performance improvements [5]. In this paper, we develop a novel push-relabel algorithm that exploits the large number of available processor cores of the GPU, and adaptively switches between the CPU and GPU during the course of the execution for optimal parallelization efficiency. To the best of our knowledge, we are not aware of any previous works that utilize the GPU for the generic maximum flow problem.

Our algorithm is built with the lock-free push-relabel technique that we previously developed in [4]. For notational convenience, we use "CUDA" and "GPU" interchangeably in the rest of the paper. Our CUDA-based algorithm is able to utilize the large number of processor cores available on a GPU and hence has the potential to outperform existing algorithms. However, due to the two limitations listed above, we observed that the speedup achieved by our CUDA-

based algorithm does not increase linearly with the number of available processor cores on a GPU. To analyze such performance discrepancy, we develop a model to describe the execution characteristics of both GPU- and CPU-based versions of the algorithm. The analysis shows that the performance degradation is due to the lack of available concurrent tasks on sparse graphs (hence many threads are wasting time idling), and also due to the noticeable overheads of exchanging data between the CPU main memory and the GPU global memory.

As the number of available concurrent tasks push-relabel algorithm varies during run time and may not be large enough to keep the GPU cores busy, we improve our algorithm to a CPU-GPU-Hybrid scheme which is able to take advantage of both serial and parallel executions. The new algorithm is based on the efficiency model that can dynamically estimate the efficiency for the current workload. An arbitrator periodically compares the efficiency of CPU and GPU and chooses the one with higher efficiency to perform the computation. Extensive experimental results show that our new algorithm is robust and efficient. Our algorithm outperforms the sequential push-relabel algorithm for all types (both dense and sparse) of input graphs that were tested. Up to 2 times improvement in execution time has been observed.

The major contributions of our paper are summarized as follows:

1) We first develop a generic parallel push-relabel algorithm for CUDA-based GPUs.
2) We then model the parallelization efficiency of our push-relabel algorithm.
3) We then develop a dynamically tuned push-relabel algorithm that can adaptively select CPU or GPU for the computation and maximize the execution efficiency.
4) We conduct extensive experiments and demonstrate the effectiveness of the proposed dynamically tuned algorithm.

The rest of the paper is organized as follows: Section II summarizes the background and related work. Section III presents our CUDA-based generic parallel push-relabel algorithm. In Section IV, we develop an efficiency model for our algorithm. The model is used in Section V to develop our dynamically tuned push-relabel algorithm. Section VI and Section VII discuss the implementation details and present the experimental results. Section VIII concludes the paper with the discussion on future research directions.

II. Background and Related Work

The max-flow problem is defined as follows: A flow network is a graph $G(V, E)$ where edge $(u, v) \in E$ has capacity c_{uv}. G has source $s \in V$ and sink $t \in V$. A flow in G is a real valued function f defined over $V \times V$ that satisfies the following constraints:

1. $f(u, v) \leq c_{uv}$ for $u, v \in V$
2. $f(v, u) = -f(u, v)$ for $u, v \in V$
3. $\sum_{v \in V} f(v, u) = 0$ for $u \in V - \{s, t\}$

The value of a flow f is defined as $|f| = \sum_{u \in V} f(s, u)$, which is the net amount of flow sent from s to t. The maximum flow problem searches for a flow with the maximum value.

Early solutions to the maximum flow problem are based on the augmenting path method due to Ford and Fulkerson [6], which by itself is 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 [7] and the $O(|V|^2|E|)$ algorithm by Dinitz [8]). The concept of preflow was introduced by Karzanov in [9], which leads to an $O(|V|^3)$ algorithm, the execution time was further improved in [10], [11]. Goldberg et al. designed the push-relabel algorithm [12] with $O(|V|^2|E|)$ operations and further improved the complexity bound by using various techniques [3].

Practical implementations of parallel algorithms have also been investigated intensively for symmetric multi-processing (SMP) and multi-core platforms. Anderson et al. [1] augmented the push-relabel algorithm with a global relabeling operation. Bader et al. [2] designed a parallel implementation using gap relabeling heuristic with considerations in the cache performance of the push-relabel algorithm. Both implementations have demonstrated good execution speed. These parallel implementations, however, share the common feature of using locks to protect every push and relabel operation *in its entirety*, which essentially sequentializes any two push/relabel operations whenever a common vertex is involved. Without lock protection, these implementations will fail to find the maximum flow. Locks are also known to have expensive overheads [13]. Parallelism in these algorithms is therefore limited by the intensive lock usages, which can lead to performance degradation especially when the number of processors scales up.

These parallel algorithms rely on locks to guarantee their correctness and hence cannot work on platforms that do not support locking primitives. CUDA-based GPUs are one such platform (although we can build locks in software using the atomic compare-and-swap function supported by CUDA, such locks are very inefficient). Therefore special techniques have been developed by researchers to design max-flow algorithms for CUDA-based GPUs. Hussein et al. [14] presented an implementation containing only push and global relabeling operations. Vineet et al. [15] reported another version which included parallel push and relabel operations, but without the global relabeling heuristic. In these methods, each push operation is divided into two phases, *push* and *pull*, to avoid lock usages. This lock-free method requires very strong synchronization and the extra *pull* operations cause substantial overheads. In addi-

tion, these existing CUDA-based max-flow algorithms are developed for computer vision applications and can only handle graphs with a grid topology that are constructed from the target images.

Our algorithm improves over the existing algorithms. By extending our previous results in [4], we utilize the atomic fetch-and-add function supported by CUDA-based GPU and integrate both push and relabel operations into the CUDA kernel. Compared with the existing parallel algorithms, our algorithm features significantly improved parallelization efficiency and the capability of handling arbitrary input graph topologies.

III. Generic Parallel Push-Relabel Algorithm on CUDA-based GPU

Before presenting our algorithm, we first briefly re-state some notations for network flow problems. Given a direct graph $G(V, E)$, function f is called a flow if it satisfies the three constraints above. Given $G(V, E)$ and flow f, the *residual capacity* $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the *residual network* of G induced by f is $G_f(V, E_f)$, where $E_f = \{(u, v)|u \in V, v \in V, c_f(u, v) > 0\}$. Thus $(u, v) \in E_f \Leftrightarrow c_f(u, v) > 0$.

For each vertex $u \in V$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into vertex u. Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But the intermediate result before an algorithm terminates may have non-zero $e(u)$'s. We say vertex $u \in V - \{s, t\}$ is *overflowing* if $e(u) > 0$. An integer valued height function $h(u)$ is also defined for every vertex $u \in V$. We say u is higher than v if $h(u) > h(v)$. We follow the definition in [12] and say h is a valid height function if $(u, v) \in E_f$ implies $h(u) \leq h(v) + 1$. If an overflowing vertex u whose height $h(u) < |V|$, we call this vertex an *active vertex*.

Maximum flow algorithms based on the push/relabel technique typically consist of two stages [12]. 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 costs $O(|E|log|V|)$ operations, which takes much less time than the first stage. For applications searching for the minimum cut such as some computer vision applications [14], [15], the second stage is not needed at all. Because the first stage dominates the execution time of the entire algorithm, our algorithm will focus on the first stage.

Our CUDA-based parallel push-relabel algorithm is presented in Algorithm 1, where the **initialize** stage, the **push_relabel_kernel** and **global_relabel_cpu** functions are defined in Programs 2-4. In the main loop of Algorithm 1, we first transfer all the necessary data into the GPU, and then launch the CUDA kernel to concurrently execute the push and relabel operations. After the kernel executes a certain number of cycles, we transfer c_f, h and e back to the CPU main memory. The CPU will perform the global relabeling by calling **global_relabel_cpu**. To detect the termination of the algorithm, a 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). If the termination condition is not satisfied, h will be transferred into CUDA again and CPU and CUDA will repeat the above procedure until $e(s) + e(t)$ becomes equal to $ExcessTotal$. When the algorithm terminates, $e(t)$ stores the value of the maximum flow.

Algorithm 1 Parallel Push-Relabel Algorithm using CUDA

1: **Initialize** e, h, c_f and $ExcessTotal$
2: copy e and c_f from the CPU main memory to the CUDA global memory
3: **while** $e(s) + e(t) < ExcessTotal$ **do**
4: copy h from the CPU main memory to the CUDA global memory
5: call **push_relabel_kernel()**
6: copy c_f, h and e from CUDA global memory to CPU main memory
7: call **global_relabel_cpu()**
8: **end while**

Our recent work [4] proposed an asynchronous lock-free push-relabel algorithm. In [4], we augmented the original push-relabel algorithm [3] by pushing excess flow to the lowest neighbor of all active vertices, which allows the push and relabel operations to be arbitrarily interleaved as long as the excessive flow and residual capacity of the edges are updated atomically. Hence we only require atomic read-modify-write operations from the target computing platform, which makes this algorithm an ideal solution for CUDA because CUDA has been supporting such atomic operations since compute capability 1.1.

In Algorithm 1, the push and relabel operations are performed by CUDA through the **push_relabel_kernel** function, which is designed using the technique we developed in [4]. In the **push_relabel_kernel** function, the algorithm 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 $h(u) > |V|$). The atomic additions and subtractions are supported by the CUDA build-in atomic operations over the global memory.

Previous studies suggested two heuristics, global relabeling and gap relabeling, to improve the practical performance

of push-relabel algorithm. The height h of a vertex helps the algorithm to identify the direction to push the flow towards the sink. Global relabeling heuristic updates the heights of the vertices with their shortest distance to the sink. This can be performed by a backward breadth-first search (BFS) from the sink in the residual graph [3]. Gap relabeling heuristic due to Cherkassky also improves the practical performance of the push-relabel method, though not as much as global relabeling does [3]. It discovers the overflowing vertices from which the sink is not reachable and then relabel these vertices to $|V|$ to avoid unnecessary further operations. Goldberg also pointed out adopting either one of global or gap relabeling can improve the practical performance of push-relabel algorithm [3].

In sequential push-relabel algorithms, global relabeling and gap relabeling are executed by the same single thread that executes the push and relabel operations. Race conditions therefore do not exist. For parallel push-relabel algorithms, the global relabeling and gap relabeling have been proposed by Anderson [1] and Bader [2] respectively. Both heuristics locks the vertices to avoid race conditions: the global or gap relabeling, push, and relabel operations are therefore pair-wise mutually exclusive. The lack of locking primitives makes such technology infeasible for our CUDA based algorithm. Hussein [14] proposed a lockstep BFS to perform parallel global relabeling, but it was shown in [14] that this design was very slow.

In our algorithm, we propose to perform global relabeling on the CPU side (instead of using CUDA) at the cost of transferring data between the global memory of CUDA to the main memory of CPU (line 6 in Algorithm 1). Although we can avoid such data transfer overheads by keep the global relabeling operation inside CUDA, this design choice loses in overall performance because global relabeling is based on BFS, which has an intrinsically sequential processing order and also relies intensively on branch instructions, for both of which CUDA is known to be very inefficient. For example, we observed that performing a global relabeling on CPU only costs 0.4 seconds on a graph of 7998000 edges (data transfer overheads included), while CUDA needs at least 2 seconds to perform the same operation on the same graph. Our algorithm therefore chooses to offload the global relabeling operation to the CPU.

Our global relabeling operation actually consists of three steps. The first step is violation-cancellation. In our lock-free algorithm [4], at a given time, the residual graph might contain some *violating edge* (u, v) for which $h(u) > h(v) + 1$. We proved in [4] that this violation edge will be canceled implicitly by a push operation from the vertex u to v. However, in Algorithm 1, these canceling push operations may not have been performed when the **push_relabel_kernel** stops. We thus need to explicitly identify and cancel such violations in the h function before performing the global relabeling operation.

The second step is the actual global relabeling operation. After finishing the global relabeling operation, we need to perform the third step, processing the excesses of the inactive vertices. Because we still need the excesses to stay at the inactive vertices for the following second stage of the algorithm, we subtract the value of those inactive excesses from the $ExcessTotal$.

In Algorithm 1, **global_relabel_cpu** is performed by the CPU periodically. The frequency of **global_relabel_cpu** can be adjusted by changing the value of KERNEL_CYCLES. After every thread finishes KERNEL_CYCLES 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_cpu** function will assign a new height for each vertex based on this topology of the residual graph (derived from c_f).

For Algorithm 1, we have the following theorem:

Theorem 1. *Given graph $G(V, E)$ with source s and sink t, Algorithm 1 finds the maximum flow with $O(|V|^2|E|)$ push and relabel operations.*

Proof Sketch: due to the CUDA support of atomic instructions in its global memory, we can show that even though the CUDA threads may execute their push and relabel operations in an arbitrarily interleaved manner, the outcome of the execution reduces to only a few simplified scenarios. By analyzing these scenarios, we can show that function f is maintained as a valid height function. A valid h guarantees that there does not exist any paths from s to t throughout the execution of the algorithm, and hence guarantees the optimality of the final solution if the algorithm terminates. The termination of the algorithm is also guaranteed by the validatity of h, as it bounds the number of push and relabel operations to $O(|V|^2|E|)$. □

Detailed proof can be referred to our results in [4]. It is not difficult to show that the CUDA specific designs of Algorithm 1 conforms to the algorithm requirements of [4]. The proof is omitted here due to the page limit.

IV. Efficiency Model for CUDA and CPU

Algorithm 1 is designed to exploit the massive parallel processing capability of CUDA-devices. It utilizes the GPU to perform the push and relabel operations that dominate the execution time, and the CPU is responsible for the intrinsically sequential global relabeling operation that the GPU is not good at. However, the improved GPU parallelism is affected by two factors: (1) the number of available push and relabel operations, and (2) the overheads of data transfer between the CPU and the GPU. In this section, we analyze the efficiency of the CUDA-based Algorithm 1 and compare it against the CPU-only implementation. The purpose of the analysis is to establish the theoretical foundation for

Program 2 Initialize e, h, c_f and $ExcessTotal$ on CPU

```
1: h(s) ← |V|
2: e(s) ← 0
3: for all u ∈ V − {s} do
4:    h(u) ← 0
5:    e(u) ← 0
6: end for
7: for all (u, v) ∈ E do
8:    c_f(u, v) ← c_uv
9:    c_f(v, u) ← c_vu
10: end for
11: for all (s, u) ∈ E do
12:    c_f(s, u) ← c_f(s, u) − c_su
13:    c_f(u, s) ← c_f(u, s) + c_su
14:    e(u) ← c_su
15:    ExcessTotal ← ExcessTotal + c_su
16: end for
```

Program 3 Implementation of **push_relabel_kernel** function for vertex u on CUDA

```
1: cycle = KERNEL_CYCLES
2: while cycle > 0 do
3:    if e(u) > 0 and h(u) < |V| 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
```

Section V that dynamically tunes the computation between the GPU and the CPU.

Let W_i denote the workload available to a computing unit (either the GPU or the CPU) at the i^{th} iteration in terms of the number of push/relabel operations. Because each active node allows exactly one push or relabel operation that needs to be performed, we approximate W_i with N_i, the number

Program 4 Implementation of **global_relabel_cpu** 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 the 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 and marked then
13:          mark u
14:          ExcessTotal ← ExcessTotal − e(u)
15:       end if
16:    end for
17: end if
```

of active vertices at the beginning of the i^{th} iteration. Let E_{cuda} denote the efficiency of CUDA-based implementation of Algorithm 1, we have

$$E_{cuda} = \frac{N_i}{T_{overhead} + \frac{N_i}{C_{cuda}}} \tag{1}$$

where $T_{overhead}$ is the time cost for overheads including CUDA kernel initialization, data transfer between the memories of CUDA and CPU, and C_{cuda} is the computing capability of the CUDA devices in terms of the number of push/relabel operations per unit of time.

On the other hand, the CPU can substitute the GPU and execute the push and relabel operations without incurring the data transfer overhead. A model similar to Equation 1 is also suitable for CPU efficiency. Because $T_{overhead}$ is negligible for the CPU, the efficiency of CPU E_{cpu} is therefore straightforward:

$$E_{cpu} = C_{cpu} \tag{2}$$

where C_{cpu} is the computing capability of the CPU, independent of N_i.

Comparing E_{cuda} and E_{cpu}, we can see that even though C_{cuda} is much greater than C_{cpu} (due to the large number of CUDA processor cores), the CUDA efficiency may still be lower than that of the CPU when $T_{overhead}$ is large or when N_i is small.

In fact, N_i is often small for sparse graphs. This is demonstrated in the following example. We sample the number of active vertices each time when global relabeling is performed. The sampling results of a Genrmf graph (details

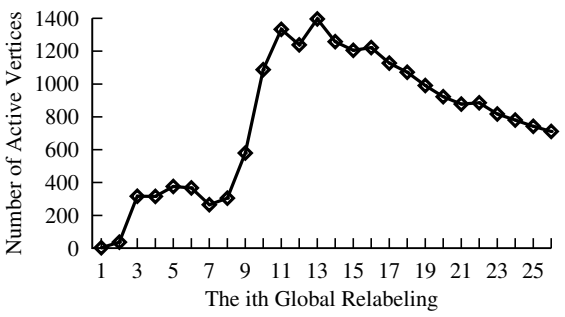

Figure 1. The number of active vertices along with the algorithm execution

of Genrmf graphs can be found in Section VII) with 262144 vertices and 1276928 edges is shown in Figure 1. We can see that though 262144 threads can be launched simultaneously, there are at most 1400 threads that have work to do. Most of the time, especially when the algorithm initially starts, only hundreds of vertices are active, which leads to a very small value of N_i.

$T_{overhead}$ is affected by a list of factors, but the most significant one is the data transfer cost. In our implementation of Algorithm 1, for each launch of the kernel, we need to transfer h, c_f and e between memories of CUDA and CPU. The launch overhead for each transfer is small (measured to be about $10\mu s$), so $T_{overhead}$ is approximately equal to $(3|V| + |E|)/B$, where B is the data transfer bandwidth between CUDA and CPU. $T_{overhead}$ for CUDA can be calculated in advance when an input graph is given.

The above analysis shows that CPU and CUDA are suitable for different workloads. In order to take advantage of both units, it is necessary to evaluate their efficiencies dynamically according to available workload. As the computation progresses, the workload should be assigned to different computing units (CUDA or CPU) to achieve optimal efficiency. By comparing Equations 1 and 2, we derive the following threshold on N_i for E_{cuda} to be higher than E_{cpu}:

$$N_i > \frac{T_{overhead} C_{cuda} C_{cpu}}{C_{cuda} - C_{cpu}} \quad (3)$$

Equation 3 shows that the threshold increases with the growth of $T_{overhead}$. When the transfer overhead is large, it is more desirable to keep all the computations on the CPU, instead of sending them to CUDA. On the other hand, when C_{cuda} is high, the computing capability of CUDA may justify the transfer overhead.

Because the actual value of C_{cpu} and C_{cuda} depends on the raw processing power of the CPU/GPU as well as the input graph (which affects vertex degree and hence the complexity of each individual push/relabel operation, as well as the data locality), there does not exist a fixed threshold value that works for all the input graphs. Actually, we observed orders of magnitudes difference in C_{cpu} and C_{cuda} when different input graphs were used. To utilize Equation 3, we propose to start the algorithm with a small threshold value and adjust it based on the dynamically measured C_{cuda} and C_{cpu}. We will elaborate the implementation details in the Section V.

Algorithm 5 Dynamically Tuned Push-Relabel Algorithm using CUDA

```
1:  Initialize e, h, c_f, ExcessTotal, aSize and Threshold
2:  LastDevice ← CPU
3:  while e(s) + e(t) < ExcessTotal do
4:    if aSize > Threshold then
5:      if LastDevice = CPU then
6:        copy e and c_f from CPU main memory to CUDA
          global memory
7:        LastDevice ← GPU
8:      end if
9:      copy h from CPU main memory to CUDA global
        memory
10:     call push_relabel_kernel()
11:     copy c_f, h and e from CUDA global memory to
        CPU main memory
12:     call global_relabel_cpu()
13:   else
14:     if LastDevice = GPU then
15:       LastDevice ← CPU
16:     end if
17:     Goldberg's serial version including global and gap
        relabeling operation (hi_pr)
18:   end if
19:   Threshold ← T_overhead C_cuda C_cpu / (C_cuda − C_cpu)
20: end while
```

V. Dynamically Tuned Push-Relabel Algorithm

In this section, we present our dynamically tuned push-relabel algorithm. The algorithm is shown in Algorithm 5. It contains not only our parallel implementation of push-relabel algorithm on CUDA (Algorithm 1), but also a CPU-based serial implementation due to Goldberg [3]. We add an arbitrator to dynamically choose, as the computation advances, the best implementation according to the comparison of their efficiencies by the model we developed in Section IV.

In Algorithm 5, the main loop of Algorithm 1 has been augmented. Before assigning the workload, we evaluate the efficiency of both CPU and CUDA in line 4 by examining the relation between two variables $aSize$ and $Threshold$ according to Equation 3. The global variable $aSize$ denotes the number of active vertices currently available. $aSize$ will be updated every time when the **global_relabel_cpu** function counts the active vertices for CUDA. We also have modified the code of Goldberg's version to track the value of $aSize$. Based on system parameters such as the data transfer

bandwidth B, the computing capability of CUDA C_{cuda} and CPU C_{cpu}, we choose an optimal threshold for $aSize$. When $aSize$ exceeds the threshold, the current workload will be assigned to CUDA; if not, the current workload will be carried on by CPU serially.

The $Threshold$ variable initially is a small value. When the algorithm starts, CPU will do all the push/relabel operations until $aSize$ reaches $Threshold$, then CUDA will be invoked to perform the push/relabel operations. We dynamically evaluate C_{cuda} and C_{cpu} by observing the number of operations that the CPU and GPU can finish in one unit of time. Next, according to Equation 3 (line 19), we derive a new value of $Threshold$. For the next iteration, the algorithm will choose either CUDA or CPU based on this new $Threshold$. $Threshold$ will be updated continuously to keep the algorithm dynamically tuned.

Compared with the CUDA-only Algorithm 1 and Goldberg's serial Algorithm [3], Algorithm 5 exhibits improved performance for all types of input graphs that were tested. The experimental results are presented in Section VII.

VI. Implementation Techniques

To evaluate the performance of our algorithms, both Algorithms 1 and 5 were implemented, and denoted as **cuda_mf** and **hybrid_mf**, respectively.

In the implementations, we allocate the edges continuously in an array which is similar to Goldberg's **hi_pr** [3]. In Goldberg's implementation, every vertex and edge has a structured entry. A vertex entry contains the excessive flow, height and an address pointer pointing to its first outgoing edge. All the outgoing edges of each vertex are allocated together, and a pointer to its mate edge (between the same vertices but in the reverse direction) is stored in its edge entry for fast reference. Besides, each edge entry stores its residual capacity and the address of its end vertex. We improve this layout for fast data exchange between CUDA and CPU by separating the pointers from the actual data items (excess, height and residual capacity). All the vertex and edge entries are divided into two parts, one stores the pointers and the other one stores the actual data. The pointers are transferred into CUDA in the initialization stage. Thus, during the execution, when we need to exchange e_f or h, we only need to transfer the actual data items. Experiments demonstrated as much as 50% reduction in the data transfer time by using this new data layout.

We further optimize the implementations by considering the specific features of CUDA. The e, h and c_f arrays are allocated in the page-locked memory by *cudaMallocHost()* that increases the speed of *cudaMemCpy* by as much as two times.

We also evaluate whether the shared memory insides CUDA should be used or not. Although CUDA hardware does not maintain coherency between the shared memory and the global memory, it supports software-based *cache emulation* which is efficient for regular data access patterns. However, the input graphs for our algorithm have various topologies and the memory access pattern is extremely irregular. We observe that utilizing the shared memory (via cache emulation) actually slows down our algorithms, i.e. the computational costs of keeping the data coherent between the shared memory and the global memory is higher than the potential benefit of the faster shared memory.

Another advantage of restricting shared memory usages is that each thread will be more independent so that we can separate the threads into as many blocks as possible. Previous implementations chose larger block sizes to allow more threads to share the same data. However, for the same number of threads, a larger block size leads to a smaller grid size. In CUDA, the threads in the same block will only be scheduled to the same multiprocessor. When the grid size is not big enough, not all the multiprocessors inside CUDA will be utilized. Using more blocks in CUDA programming therefore often leads to better performance.

Consequently, we implemented our algorithms without using the shared memory, which allows block sizes as small as 32. This enables the utilization of all the 30 multiprocessors of our experimental CUDA devices, even for graphs as small as 1000 vertices.

VII. Experimental Results

In this section, we present the experimental results to demonstrate the efficiency of our algorithms. We compare the performance of **cuda_mf** and **hybrid_mf** against Golgberg's serial implementation **hi_pr**, which is a sequential push-relabel algorithm with both global relabeling and gap relabeling. In **hi_pr**, vertices are processed in the descending order of their heights, which leads to the complexity of $O(|V|^2\sqrt{|E|})$ [16]. This is currently the fastest sequential implementation of push-relabel algorithm that we are aware of. It is also faster than the two existing parallel push-relabel algorithms [1], [2] that are designed for SMP and multi-core platforms.

The experimental platform consists of a 3.0GHz AMD quad-core Phenom II 940 processor and an Nvidia GTX 285 graphic card with 1G on-board memory. GTX 285 has a total number of 30 multiprocessors, each of which has 8 cores. The nvcc release 2.3 and Linux kernel version 2.6.27 were used for the experiments.

We tested five types of input graphs used in the 1^{st} DIMACS Implementation Challenge [17]:

1) **Acyclic-Dense graphs**: These are complete directed acyclic dense graphs: each vertex is connected to every other vertex. We tested graphs of 2000, 4000 and 6000 vertices.
2) **Genrmf graphs**: These graphs are comprised of l_1 square grids of vertices (frames) each having $l_2 \times l_2$ 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 $l_1 = 192, l_2 = 24$ (110592 vertices and 533952 edges), $l_1 = 224, l_2 = 328$ (175616 vertices and 852208 edges) and $l_1 = 256, l_2 = 32$ (262144 vertices and 1276928 edges).

3) **Genrmf-wide graphs**: The topology is the same as genrmf-long graphs except for the values of l_1 and l_2. Frames are bigger in Genrmp-wide graphs than in Genrmp-long graphs. We tested the graphs of $l_1 = 36, l_2 = 36$ (46656 vertices and 226800 edges), $l_1 = 48, l_2 = 48$ (110592 vertices and 541440 edges). $l_1 = 64, l_2 = 64$ (262144 vertices and 1290240 edges).
4) **Washington-RLG-long graphs**: These graphs are rectangular grids of vertices with w rows and l columns. Every 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 $w = 512, l = 1024$ (524290 vertices and 1572352 edges), $w = 768, l = 1280$ (983042 vertices and 2948352 edges) and $w = 1024, l = 1536$ (1572866 vertices and 4717568 edges).
5) **Washington-RLG-wide graphs**: Same as Washington-RLG-long graphs except for the values of w and l. Each row in the Washington-RLG-wide graphs are wider. We tested the graphs of $w = 512, l = 512$ (262146 vertices and 785920 edges), $w = 768, l = 768$ (589826 vertices and 1768704 edges) and $w = 1024, l = 1024$ (1048578 vertices and 3144704 edges)

For each type of graphs, 50 instances were generated using different seeds for the pseudo-random generator. Every instance was tested 3 times for each algorithm.

During the experiments, we observed that a fixed value of KERNEL_CYCLES achieves the best performance. We set the KERNEL_CYCLES to 32 for Acyclic-Dense graphs and 4096 for Genrmf graphs and Washington-RLG graphs. The execution results reported for each program were the averages over the 150 runs.

We first tested the impact of the data transfer bandwidth between CUDA and CPU. Equation 1 shows that the bandwidth influences $T_{overhead}$ and subsequently the overall CUDA performance. By adjusting the PCI-E settings, we configured bandwidth from 2.4 GBytes/sec to 0.6 GBytes/sec. 50 Washtington-RLG-long graphs (983042 vertices and 2948352 edges) were used for this set of experiments. The results are shown in Figure 2. Because **hi_pr** does not transfer data through the PCI bus, it is not affected by changes in the bandwidth. But **cuda_mf** and **hybrid_mf** both suffered from the decrease of the bandwidth. When the bandwidth reduced from 2.4 GBytes/sec to 0.6 GBytes/sec, **cuda_mf** lost 30% in its performance. Our **hybrid_mf** was able to dynamically adjust the workload assignment so that it suffered much less than **cuda_mf** with almost negligible degradation in performance. In the following experiments, we set the PCI-E bandwidth to 2.4 GBytes/sec.

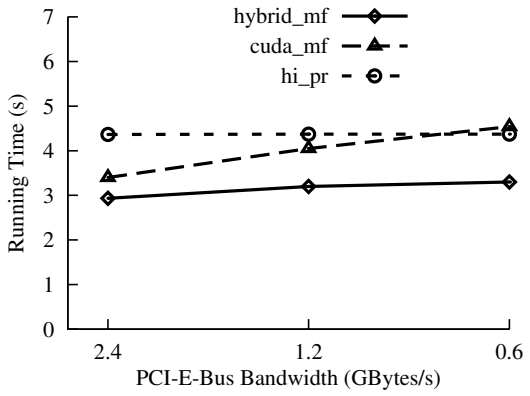

Figure 2. The bandwidth impacts on cuda_mf, hybrid_mf and hi_pr

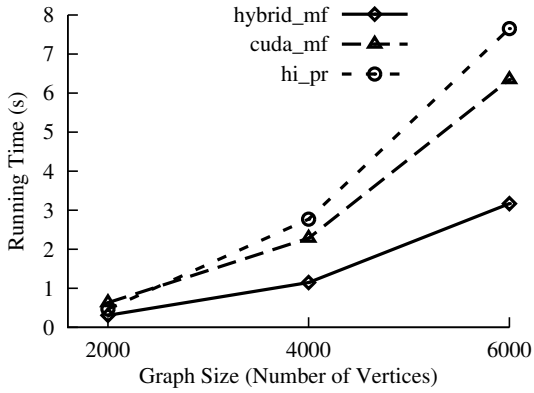

Figure 3. The performance comparison among cuda_mf, hybrid_mf and hi_pr on Acyclic-Dense graphs

Figure 3 shows the results on Acyclic-Dense graphs. We can observe that **hi_pr** slightly outperforms **cuda_mf** for graphs of size 2000, but when the graph size increases, the performance of **cuda_mf** exceeds that of **hi_pr**, because the number of working threads on CUDA increases when the graph size increases. Nevertheless, **hybrid_mf** is faster than both **cuda_mf** and **hi_pr** for all graph sizes from 2000 to 6000. When the graph size is 6000, **hybrid_mf** is faster than **hi_pr** by 2 times. We observed that **hybrid_mf** usually launched the **push_relabel_kernel** only twice (compared with **cuda_mf**'s over ten times), and the rest of work is completed by the CPU. The experiments demonstrated that **hybrid_mf** can take advantages of both the CPU and CUDA so that it outperforms both.

We also observed similar results on Washington-RLG

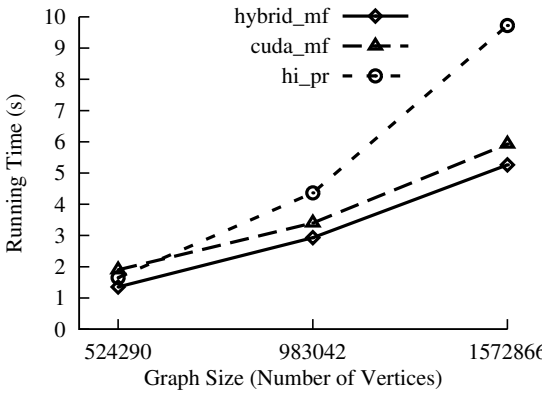

Figure 4. The performance comparison among cuda_mf, hybrid_mf and hi_pr on Washington-RLG-long graphs

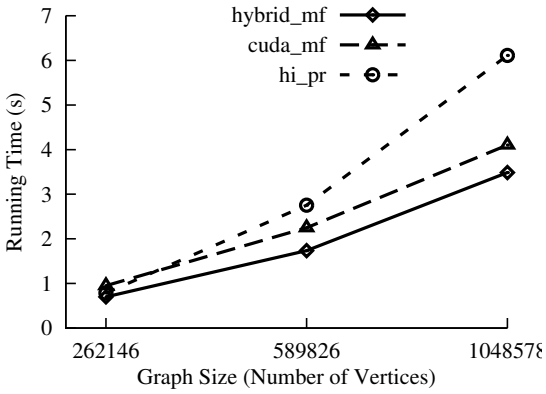

Figure 5. The performance comparison among cuda_mf, hybrid_mf and hi_pr on Washington-RLG-wide graphs

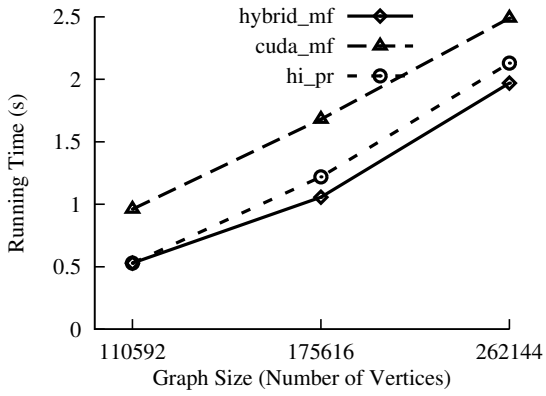

Figure 6. The performance comparison among cuda_mf, hybrid_mf and hi_pr on Genrmf-long graphs

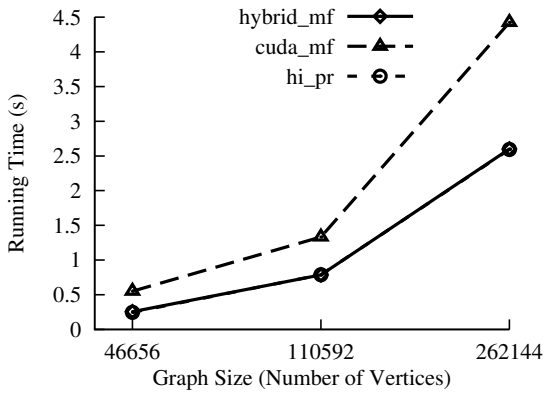

Figure 7. The performance comparison among cuda_mf, hybrid_mf and hi_pr on Genrmf-wide graphs

graphs as shown in Figures 4 and 5.

On Genrmf graphs shown in Figure 6 and 7, **cuda_mf** is much slower than **hi_pr**, but **hybrid_mf** is still faster than **hi_pr**. Genrmf graphs are typical sparse graphs, which has $\sim$110,000 vertices with an average vertex degree of 5. We observed in the experiments that less than 1% of the vertices are typically active during the course of the computation. For this type of graphs, CUDA is not as efficient as CPU. **hybrid_mf** adaptively chooses CPU to perform most of the operations and thus exhibits very close performance to **hi_pr**

In summary, the experiments show that **cuda_mf** and **hi_pr** are suitable for different scenarios. Our dynamically tuned algorithm **hybrid_mf** is able to take advantage of both **cuda_mf** and **hi_pr**. By utilizing the efficiency model, **hybrid_mf** has achieved the highest efficiency among all these three implementations.

VIII. Conclusion and Discussions

In this paper, we presented two push-relabel algorithms using CUDA. The first algorithm uses CUDA to conduct all the push/relabel operations and uses the CPU to perform global relabeling operation. The complexity of the algorithm is $O(|V|^2|E|)$. We further improve the performance of this CUDA-only algorithm with a hybrid algorithm that adaptively chooses CUDA and CPU to conduct the push and relabel operations. Guided by our efficiency model, this algorithm is able to dynamically assign the workload to the computing unit with higher computing efficiency. Intensive experiments show that the adaptive algorithm outperforms both the CUDA-only version and Goldberg's sequential algorithm by as much as 2 times.

The algorithm can be improved in the following aspects and we plan to extend our research along these directions. First, $T_{overhead}$, the overhead of data transfer can be reduced by managing the volume of the data exchanging. Secondly, we plan to investigate the enforcement of an approximate highest-label-first ordering on the push and relabel operations conducted by CUDA. The order is expected to reduce the total number of operations and improve the overall execution speed of the algorithm.

ACKNOWLEDGMENT

This work is supported by the US National Science Foundation under award number CNS-0845583.

REFERENCES

[1] R. J. Anderson and a. C. S. Jo "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*. New York, NY, USA: ACM, 1992, pp. 168–177.

[2] D. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *PDCS '05: Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

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

[4] B. Hong, "A lock-free multi-threaded algorithm for the maximum flow problem," in *IEEE International Parallel and Distributed Processing Symposium*, Aprail 2008.

[5] S. Lahabar and P. J. Narayanan, "Singular value decomposition on gpu using cuda," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.

[6] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[7] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[8] E. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Mathematics Doklady*, vol. 11, pp. 1277–1280, 1970.

[9] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Mathematics Doklady*, vol. 15, pp. 434–437, 1974.

[10] H. N. Gabow, "Scaling algorithms for network problems," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 148–168, 1985.

[11] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by successive approximation," *Math. Oper. Res.*, vol. 15, no. 3, pp. 430–466, 1990.

[12] ——, "A new approach to the maximum flow problem," in *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1986, pp. 136–146.

[13] D. Culler, J. P. Singh, and A. Gupta., *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[14] M. Hussein, A. Varshney, and L. Davis, "On implementing graph cuts on cuda," in *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[15] V. Vineet and P. Narayanan, "Cuda cuts: Fast graph cuts on the gpu," 2008, pp. 1–8.

[16] J. Cheriyan and K. Mehlhorn, "An analysis of the highest-level selection rule in the preflow-push max-flow algorithm," *Information Processing Letters*, vol. 69, pp. 69–239, 1998.

[17] D. S. Johnson and C. C. McGeoch, Eds., *Network Flows and Matching: First DIMACS Implementation Challenge*. Boston, MA, USA: American Mathematical Society, 1993.