A Push-Relabel-Based Maximum Cardinality Bipartite Matching Algorithm on GPUs

Mehmet Deveci, Kamer Kaya, Bora Uçar, Umit V. Catalyurek

A Push-Relabel-Based Maximum Cardinality Bipartite Matching Algorithm on GPUs

Mehmet Deveci[1,2], Kamer Kaya[1], Bora Uçar[4], Ümit V. Çatalyürek[1,3]
[1]Dept. of Biomedical Informatics
[2]Dept. of Computer Science and Engineering
[3]Dept. of Electrical and Computer Engineering
The Ohio State University
[4]CNRS and LIP, ENS Lyon, France
{mdeveci,kamer,umit}@bmi.osu.edu
bora.ucar@ens-lyon.fr

***Abstract*—We design, develop, and evaluate an atomic- and lock-free GPU implementation of the push-relabel algorithm in the context of finding maximum cardinality matchings in bipartite graphs. The problem has applications on computer science, scientific computing, bioinformatics, and other areas. Although the GPU parallelization of the push-relabel technique has been investigated in the context of flow algorithms, to the best of our knowledge, ours is the first study which focuses on the maximum cardinality matching. We compare the proposed algorithms with serial, multicore, and manycore bipartite graph matching implementations from the literature on a large set of real-life problems. Our experiments show that the proposed push-relabel-based GPU algorithm is faster than the existing parallel and sequential implementations.**

***Index Terms*—Push-relabel, GPU, maximum cardinality matchings, bipartite graphs**

I. Introduction

Bipartite graph matching is a classical problem in graph theory and combinatorial optimization. Given a bipartite graph, we want to find a set of vertex disjoint edges with the maximum cardinality. The problem is of interest in a variety of fields such as bioinformatics [**?**], scheduling [**?**, Section 3.8], and image processing [**?**]. Maximum cardinality bipartite matching is also employed routinely in sparse linear solvers to see if the associated coefficient matrix is reducible; if so, substantial savings in computational requirements can be achieved [**?**, Chapter 6].

Different approaches for computing maximum matchings in bipartite graphs exist in the literature. Duff et al. [**?**] discuss the design, analysis and sequential implementation of a class of algorithms which are based on augmenting paths. Push-relabel-based algorithms [**?**] form the second class. Their design and careful sequential implementation have been recently investigated. A rigorous set of experiments showed that the push-relabel approach is very promising and performs as good as the augmenting-path-based approach [**?**]. The implementation of the third, more recent, approach based on pseudoflow algorithms [**?**], have been described by Chandran and Hochbaum [**?**].

Algorithms dealing with graph problems are usually memory bounded, hence it is relatively hard to obtain a good parallel performance. Moreover, because of the computation irregularities, it is difficult to exploit concurrency. The matching problem is no exception. There have been recent studies that aim to improve the performance of matching algorithms on multicore and manycore architectures. For example, Vasconcelos and Rosenhahn [**?**] propose a GPU implementation of an algorithm for the maximum weighted matching problem on bipartite graphs. Fagginger Auer and Bisseling [**?**] study an implementation of a greedy graph matching on GPU. Çatalyürek et al. [**?**] propose different greedy graph matching algorithms for multicore architectures. Azad et al. [**?**] introduce several multicore implementations of the maximum cardinality matching algorithms on bipartite graphs. In a recent study [**?**], we investigated the performance of augmenting-path-based approach on GPU architectures.

In this work, we investigate the GPU parallelization of the push-relabel-based maximum cardinality bipartite matching algorithm on GPUs. We develop an atomic- and lock-free implementation to obtain a good performance. Our approach employs the global and gap relabeling heuristics which are also implemented in the GPU. We thoroughly evaluate the performance of the algorithm with a rigorous set of experiments on many bipartite graphs from different applications. The experimental results conclude that the proposed GPU-based implementation is faster than the existing sequential, multicore, and manycore approaches.

The rest of this paper is organized as follows. The background material, some related work, and a summary of contributions are presented in Section **??**. Section **??** describes the proposed GPU algorithm and the techniques we develop to make it faster. The comparison of the proposed algorithm with its existing sequential, multicore, and manycore counterparts is given in Section **??**. Section **??** concludes the paper.

II. Background and Related Work

In a bipartite graph $G = (V_1 \cup V_2, E)$ with two disjoint vertex sets V_1 and V_2, each edge in E has one endpoint in V_1 and one in V_2. Let m and n be the number of vertices in V_1 and V_2 respectively. And let $\Gamma(v) = \{u : \{u, v\} \in E\}$ represent the *neighborhood* of a vertex $v \in V_1 \cup V_2$. To better distinguish

the vertices in V_1 and V_2 in the text, we will employ the matrix notation and use V_R and V_C for V_1 and V_2, respectively, and refer the vertices in V_R as the set of rows and the vertices in V_C as the set of columns.

A subset of the edges, $\mathcal{M}$, is called a *matching* if no two edges in $\mathcal{M}$ share an endpoint. If $v \in V$ is an endpoint in $\mathcal{M}$ we say v is *matched*. Otherwise, it is *unmatched*. A matching $\mathcal{M}$ is called *maximal*, if there exists no other matching $\mathcal{M}'$ such that $\mathcal{M}' \supset \mathcal{M}$. A maximal matching $\mathcal{M}$ is called *maximum* if $|\mathcal{M}| \geq |\mathcal{M}'|$ for all possible matchings $\mathcal{M}'$ where $|\mathcal{M}|$ is the cardinality of $\mathcal{M}$. If $|\mathcal{M}| = |V_R| = |V_C|$, $\mathcal{M}$ is called *perfect matching*. The *deficiency* of a matching $\mathcal{M}$ is the difference between the cardinality of a maximum matching and $|\mathcal{M}|$.

A. Maximum cardinality bipartite matching algorithms

A path in G is $\mathcal{M}$-*alternating* if its edges alternate between those in matching $\mathcal{M}$ and those not in $\mathcal{M}$. An $\mathcal{M}$-alternating path $\mathcal{P}$ is called $\mathcal{M}$-*augmenting* if the start and end vertices of $\mathcal{P}$ are both unmatched. Augmenting path-based algorithms are based on the following theorem.

*Theorem 1 ([**?**]):* Let G be a graph (bipartite or not) and let $\mathcal{M}$ be a matching in G. Then $\mathcal{M}$ is of maximum cardinality if and only if there is no $\mathcal{M}$-augmenting path in G.

Given a possibly empty matching $\mathcal{M}$, augmenting-path-based algorithms searches for an $\mathcal{M}$-augmenting path $\mathcal{P}$. If none exists then the matching $\mathcal{M}$ is maximum by Theorem **??**. Otherwise, $\mathcal{P}$ is used to increase the cardinality of $\mathcal{M}$ by setting $\mathcal{M} = \mathcal{M} \oplus E(\mathcal{P})$ where $E(\mathcal{P})$ is the edge set of a path $\mathcal{P}$, and $M \oplus E(\mathcal{P}) = (\mathcal{M} \cup E(\mathcal{P})) \setminus (\mathcal{M} \cap E(\mathcal{P}))$ is the symmetric difference. Since both the first and the last edge of $\mathcal{P}$ were unmatched in $\mathcal{M}$, we have $|\mathcal{M} \oplus E(\mathcal{P})| = |\mathcal{M}| + 1$. The way in which augmenting paths are searched constitutes the main difference between the algorithms based on augmenting paths, both in theory and in practice. These algorithms mainly use graph traversal techniques such as the depth-first search (DFS) and breadth-first search (BFS). We refer the reader to [**?**] for a recent survey on these algorithms.

B. The push-relabel algorithm for bipartite matching

Push-relabel algorithms search and augment simultaneously. They do not explicitly construct augmenting paths but they follow them with a clever heuristic. That is, they repeatedly augment a prefix of a hypothetical augmenting path $\mathcal{P}_2 = (v, u, w)$ in G where $w \in V_C$ is matched to $u \in V_R$, and $v \in V_C$ is an unmatched column. Augmentations are performed by unmatching w and matching v to u. If w has an unmatched neighbor, the suffix of an augmenting path has been found, allowing the augmentation of $|\mathcal{M}|$. These hypothetical augmentation operations are performed until no further suffixes can be found. The algorithms are guided by assigning a label to every vertex which provides a lower bound on the distance to the nearest unmatched row.

Goldberg and Tarjan's original push-relabel algorithm was designed for the maximum flow problem [**?**]. The bipartite matching problem is a special case of maximum flow. Hence, the algorithm can be simplified to be adapted to the bipartite matching problem. In fact, it is known to be one of the fastest algorithms for bipartite matching [**?**]. In the following, we portray this algorithm in a ready-to-implement pseudocode form as shown in Algorithm **??**. This algorithm will be referred to as PR throughout the paper.

Algorithm 1 PR: Push-Relabel based Bipartite Matching

Input: A bipartite graph $G = (V_R \cup V_C, E)$ and a (possibly empty) matching $\mathcal{M}$
Output: A maximum cardinality matching $\mathcal{M}^\star$

```
1: Set ψ(u) = 0 for all u ∈ V_R
2: Set ψ(v) = 1 for all v ∈ V_C
3: Set all v ∈ V_C unmatched by M to active
4: while an active column v exists do
5:    Find a row u ∈ Γ(v) of minimum ψ(u)
6:    if ψ(u) < m + n then
7:       if {u, w} ∈ M then
8:          M ← M \ {u, w}/*Single push*/
9:          Set w active
10:      M ← M ∪ {u, v}  /*Double push*/
11:      ψ(v) ← ψ(u) + 1   /*Relabels v*/
12:      ψ(u) ← ψ(u) + 2 /*Relabels u*/
13:   Set v inactive
14: return M* = M
```

Let $\psi(v)$ denote a lower bound on the length of the shortest alternating-path from the vertex v to an unmatched row. Hence, ψ can be used to find the direction of the closest unmatched row for each vertex. If v is an unmatched column, this path is also an augmenting path. During the initialization of PR, the algorithm sets $\psi(v) = 1$ for all $v \in V_C$ and $\psi(v) = 0$ for all $v \in V_R$. Throughout the algorithm, the unmatched columns are called *active*.

As long as an active column v exists, the algorithm performs the *push* operation. $\Gamma(v)$ is searched for a row $u \in \Gamma(v)$ with the minimum $\psi(u)$. Note that $\psi(v) - 1$ is the infimum for the value of $\psi(u)$. This property, $\psi(u) \geq \psi(v) - 1$ for u with the minimum ψ in the neighborhood of v, holds after the initialization ($\psi(u) = 0$ and $\psi(v) = 1$ for all $u \in V_R$ and $v \in V_C$) and is maintained throughout the algorithm as the neighborhood invariant. Hence, when an edge $\{v, u\}$ with $\psi(v) = \psi(u) + 1$ is found the search stops.

Let u be the row with minimum ψ in $\Gamma(v)$. If u is unmatched, it can be matched to v, and the cardinality of $\mathcal{M}$ can be increased by one (*single push*). Note that an unmatched row vertex u has $\psi(u) = 0$, and it will always have the minimum ψ value. If u is matched to a column w, a *double push* operation is performed which removes $\{w, u\}$ from $\mathcal{M}$, adds $\{v, u\}$ to $\mathcal{M}$, and makes w active. Hence, once a row is matched, it never becomes unmatched again.

After the push operation, PR relabels v with $\psi(v) \leftarrow \psi(u) + 1$. By definition, $\psi(u)$ is a lower bound on the alternating-path distance from u to a closest unmatched row. Since the path between v and its closest unmatched row must contain some $u' \in \Gamma(v)$, and $\psi(u)$ has the minimum ψ among all the neighbors of v, $\psi(u) + 1$ is a lower bound on the length of a path between v and its closest unmatched row.

After relabeling v, PR relabels u by $\psi(u) \leftarrow \psi(u) + 2$. For a single push, $\psi(u)$ becomes 2. For a double push, any alternating path from u to a closest unmatched row now contains v. Such a path always uses the edge (u, v), and hence, $\psi(u)$ must be at least $\psi(v) + 1$ which is actually $\psi(u) + 2$ due to relabeling of v. Clearly, this new value maintains the neighborhood invariant $\psi(u) \geq \psi(v) - 1$.

The maximum length of any augmenting path in G is at most $\min(2m, 2n) - 1$. Since ψ is a lower bound on the length of a path to an unmatched row, when the minimum $\psi(u)$ among all $u \in \Gamma(v)$ becomes $m+n$, the vertex v cannot be on an augmenting path. In this case, no push operations are realized and v is considered unmatchable, i.e., *inactive*. The push and relabel operations are repeated until there is no active column left, either because all vertices have been matched or marked as inactive. Using Theorem **??**, it is easy to show that in this case $\mathcal{M}$ is a maximum matching. The time complexity of the algorithm is $\mathcal{O}((n + m)\tau)$ [**?**].

A push-relabel implementation does not need to store a $\psi(u)$ for a row u since the value is 0 when u is unmatched and $\psi(w) + 1$ when $(u, v) \in \mathcal{M}$. However, such an approach increases the amount of jumps and arithmetic operations. Thus, we store a full ψ array for both the rows and the columns.

C. Global relabeling

As various studies showed, PR's performance can be improved by periodically setting all labels to exact distances. This heuristic is called *global relabeling* and realized via a BFS starting from all unmatched rows as shown in Algorithm **??**. During the traversal, the label of each vertex v is set to the minimum distance from v to any unmatched row. And each vertex w not visited by the BFS is assigned a label $\psi(w) = m+n$, hence it is removed from further consideration.

Algorithm 2 GR: Global Relabeling

Input: A bipartite graph $G = (V_R \cup V_C, E)$ and a matching $\mathcal{M}$ in G
Output: An accurate distance labeling ψ w.r.t. $\mathcal{M}$

```
1: Q ← u for all unmatched u ∈ V_R
2: Set ψ(v) = m + n for all v ∈ V_C
3: Set ψ(u) = m + n for all matched u ∈ V_R
4: while Q not empty do
5:     u ← POP u from Q
6:     for all v ∈ Γ(u) do
7:         if ψ(v) = m + n then
8:             ψ(v) ← ψ(u) + 1
9:             if {v, w} ∈ M then
10:                ψ(w) ← ψ(v) + 1
11:                PUSH w to Q
12: return ψ
```

In practice, a counter is incremented every time the value of $\psi(v)$ is changed in Line **??** of Algorithm **??** in order to keep track of the number of pushes. The usual approach is to call GR when this counter reaches a predetermined threshold. A threshold of n was suggested as the standard frequency of global relabels [**?**]. The performance of the push-relabel approach and its several sequential implementations for the bipartite cardinality matching problem have been well studied and various improvements have been proposed [**?**], [**?**], [**?**]. For example, it has been shown that when the active columns are visited with a FIFO order, the push-relabel approach obtains a better performance. The most recent implementation and a comparison with the augmenting-path-based algorithms are given by Kaya et al. [**?**]. The authors showed that PR is slightly better than the augmenting-path-based algorithms. In our experiments, we use their implementation for comparison purposes.

D. Parallel algorithms for bipartite cardinality matchings

Efficient multicore counterparts of a number of augmenting-path based algorithms have been proposed in a recent study [**?**]. The parallelization of the multicore algorithms is achieved by using atomic operations at BFS and/or DFS steps of the algorithm. Among these algorithms, P-DBFS, which employs vertex disjoint BFSs to find the augmenting paths, obtained the best performance in our experiments. Its relative performance was also good in the original experiments [**?**]. Thus, in this paper, we compare the performance of our GPU-based push relabel algorithm with P-DBFS.

Although using atomic operations might not harm the performance on a multicore machine, they should be avoided in a GPU implementation because of very large number of concurrent thread executions. In recent work, we designed and developed atomic-free GPU implementations, G-HK and G-HKDW [**?**], corresponding to two augmenting-path-based algorithms, HK [**?**] and HKDW [**?**]. HK has the best known worst-case running time complexity of $O(\tau\sqrt{n+m})$ for a bipartite graph with τ edges. HKDW is a variant of HK and incorporates techniques to improve the practical running time while having the same worst-case time complexity. Both of these algorithms use BFS to locate the shortest augmenting paths from unmatched columns, and then use DFS-based searches restricted to a certain part of the input graph to augment along a maximal set of disjoint augmenting paths. HKDW performs another set of DFS-based searches to augment using the remaining unmatched rows. In this paper, we compare the performance of the proposed algorithm with that of G-HKDW which performed better than G-HK [**?**].

GPU-based push-relabel implementations for the maximum flow problem exist in the literature [**?**], [**?**], [**?**]. Vineet and Narayanan used the push-relabel approach to find graph cuts in GPUs [**?**]. Their implementation does not have the global relabeling heuristic which is later added by Huseein et al. [**?**]. However, the implementation does not have a relabeling operation except the global one. He and Hong proposed a GPU-based implementation of the push-relabel approach with global relabeling on hybrid CPU-GPU architectures for the maximum flow problem [**?**]. To integrate both push and relabel operations into a CUDA kernel, they used the atomic fetch-and-add function supported by CUDA-based GPUs.

To the best of our knowledge, ours is the first GPU implementation of the push-relabel approach for the maximum bipartite cardinality matching problem. The algorithm does not

need locks or atomic instructions to perform push-relabel operations concurrently. Instead, the algorithm allows matching inconsistencies throughout its execution, which are resolved at the end. Furthermore, it is implemented fully on the GPU with CUDA including a global relabeling phase.

III. Push-Relabel-based Bipartite Cardinality Matching on GPUs

A high-level structure of our GPU-based algorithm is given in Algorithm **??**. The main **while** loop at line **??** iterates until all active columns are consumed by the GPU kernel executions at line **??** where in a single execution of G-PR-Krnl, an active column is processed by a single GPU thread. In our implementation, we use an array $\psi(\cdot)$ to store the labels of vertices in $V_R \cup V_C$ and another array $\mu(\cdot)$ to store their matching vertices. During the course of the algorithm, if two vertices $u \in V_R$ and $v \in V_C$ are matched $\mu(u) = v$ and $\mu(v) = u$. If a row $u \in V_R$ is unmatched, $\mu(u) = -1$. If a column $v \in V_C$ is inactive, either $\mu(v) = -2$, i.e., v cannot reach to an unmatched row with an augmenting path, or $\mu(v) = u$ and $\mu(u) = v$, i.e., v is matched with a $u \in V_R$. The algorithm maintains these as its matching invariants throughout the execution. Note that a column vertex v can have $\mu(v) > -1$ even though it is unmatched.

Algorithm 3 G-PR: GPU-based Push-Relabel Matching

Input: $G = (V_R \cup V_C, E)$, μ
1: $loop \leftarrow 0$
2: $actExists \leftarrow$ **true**
3: $iterGR \leftarrow 0$
4: **while** $actExists$ **do**
5: **if** $loop = iterGR$ **then**
6: $\langle maxLevel, \psi \rangle \leftarrow$ G-GR(G, μ)
7: $iterGR \leftarrow$ GetIterGR$(maxLevel, loop, iterGR)$
8: $actExists \leftarrow$ **false**
9: $\langle \mu, \psi, actExists \rangle \leftarrow$ G-PR-Krnl(G, μ, ψ)
10: $loop \leftarrow loop + 1$
11: $\mu \leftarrow$ FixMatching(μ)
12: **return** μ

A. Global relabel operation on GPUs

The proposed algorithm G-PR periodically uses a global-relabeling function to set exact ψ values. The pseudocode of this function is given in Algorithm **??**. The function first calls a GPU kernel InitRelabel. For each unmatched row u, the initialization kernel sets $\psi(u) \leftarrow 0$. For all other vertices, ψ values are set to $m + n$. After the initialization, a global relabeling is performed by multiple G-GR-Krnl calls where each call processes a level of a BFS starting from all unmatched rows. The details of the kernel is given in Algorithm **??**. The row vertices are distributed to the threads and if a row u has $\psi(u) = cLevel$ all of the unvisited columns $v \in \Gamma(u)$ are visited. For each such column v, $\psi(v)$ is set to $cLevel+1$ and its matched row's ψ value is set to $cLevel+2$. Although the same $\psi(\cdot)$ value can be set by multiple threads, since all threads use the same value for the same location there is no concurrency issues for G-GR-Krnl.

Algorithm 4 G-GR: GPU-based Global Relabeling

Input: $G = (V_R \cup V_C, E), \mu$
1: $\psi \leftarrow$ InitRelabel(μ)
2: $uAdded \leftarrow$ **true**
3: $cLevel \leftarrow 0$
4: **while** $uAdded$ **do**
5: $uAdded \leftarrow$ **false**
6: $\langle \psi, uAdded \rangle \leftarrow$ G-GR-Krnl $(G, \psi, \mu, cLevel)$
7: $cLevel \leftarrow cLevel + 2$
8: $maxLevel \leftarrow cLevel$

Algorithm 5 G-GR-Krnl

Input: $G = (V_R \cup V_C, E), \psi, \mu, cLevel$
1: **for all** row vertex $u \in V_R$ **in parallel do**
2: **if** $\psi(u) = cLevel$ **then**
3: **for all** $v \in \Gamma(u)$ **do**
4: **if** $\psi(v) = m + n$ **then**
5: $\psi(v) \leftarrow cLevel + 1$
6: **if** $\mu(v) > -1$ **and** $\mu(\mu(v)) = v$ **then**
7: $\psi(\mu(v)) \leftarrow cLevel + 2$
8: $uAdded \leftarrow$ **true**

Following the practice, in our experiments, we used an initial matching $\mathcal{M}$ obtained by a standard greedy matching heuristic [**?**]. When the initial matching is empty, we start with $\psi(u) = 0$ for each $u \in V_R$, and $\psi(v) = 1$ for each $v \in V_C$. Even with a nonempty matching, the algorithm runs correctly after this initialization. However, our preliminary experiments show that applying a global relabeling at the beginning of the main while loop of G-PR leads significant performance improvements. Hence, we set $iterGR$ to 0 (line **??** of Algorithm **??**) which stores the iteration number in which the next global relabeling will be performed. Experiments in the literature show that the performance of the algorithm changes drastically with the strategy to set $iterGR$ [**?**], [**?**]. In our preliminary experiments, we also observed this sensitivity with different implementations of GetIterGR (line **??** of Algorithm **??**). In practice, a global relabeling is performed after every $k \times (n + m)$ pushes where the suggested k values are between 1 and 2 [**?**], [**?**]. Unlike the sequential push-relabel algorithms, it is very expensive to count the number of pushes performed during kernel executions. Hence, for the GPU-based implementation, we need a different strategy. We initially experimented with fixed intervals and performed a global relabeling after every k iterations of the main while loop. We then developed an adaptive strategy where the next global relabeling is done after $k \times maxLevel$ loop iterations and experimented with several k values. The design rationale behind this strategy is based on the following theorem:

*Theorem 2 ([**?**]):* If $\mathcal{M}$ is a matching with a deficiency d there exists a set of d vertex disjoint augmenting paths.

Let $\mathcal{S}$ be the set of paths in Theorem **??**. Since the paths are vertex disjoint, their total length can be used as a lower bound for $m+n$. Since the BFS in a global relabeling uses alternating paths and these paths stop at unmatched columns, a fraction of the $maxLevel$ is a reasonable estimate on the average path length in $\mathcal{S}$. Thus, following the sequential practice, using $k \times$

$maxLevel$ kernel executions with d active columns can be promising. Our experiments show that the adaptive strategy is superior to global relabeling with fixed intervals.

B. Push relabel operations on GPUs

After a global relabeling in Algorithm **??**, the kernel G-PR-KRNL processes the active columns and performs concurrent push-relabel operations. The first implementation of this kernel is given in Algorithm **??**. As described in Section **??**, there are three cases while processing an active column $v \in V_C$: Let $u \in V_R$ be the row in $\Gamma(u)$ with a minimum ψ value. If $\psi(u) < m+n$ a single push matches v with u and consumes it (lines **??** and **??**). If $\mu(u)$ was previously matched with a $v' \in V_C$ a double push produces a new active column v'. Note that we do not actually implement the double-push operation and fix $\mu(v')$ since it has never been solely used, i.e., we always check $\mu(\mu(v'))$ while we want to know if the column is matched or unmatched. If $\psi(u) = m+n$ then v is set to inactive and consumed (line **??**).

Concurrent push-relabel operations in a single G-PR-KRNL execution do not create a problem since when two active columns v and v' perform pushes simultaneously with a row u, we have $\mu(v) = \mu(v') = u$ and $\mu(u) \in \{v, v'\}$. Hence, although one of the $\mu(v)$ and $\mu(v')$ is wrong, the inconsistency in the matching can be captured by the algorithm. A problem might arise if there is an inconsistency in the neighborhood and matching invariants of the algorithm. We claim that the invariants are not violated neither. If both v and v' select u at the same time for the push operation, they will both store the same $\psi(u)$ as ψ_{min}. If v completes the push operation earlier than v', first $\psi(v)$ and $\psi(u)$ will be updated for v. Although the matching of v will be invalidated by v', the update of $\psi(v)$ does not violate the neighborhood invariant as $\psi_{min} = \psi(u) \geq \psi(v) - 1$. Therefore, although v has another neighbor u' with $\psi(u') = \psi_{min}$, both invariants will still be valid. The double update of $\psi(u)$ does not create a problem either, as both of the threads set it to the same value. A $\psi(u)$ entry can be set more than once with different values within a single execution of G-PR-KRNL. If this is the case, these push operations can be considered as two consecutive pushes in a sequential execution as they have different ψ_{min} values stored at the beginning of push operation.

The while loop of the main algorithm G-PR iterates and continues to call the kernel G-PR-KRNL until all the active columns are consumed. Once the maximum cardinality matching is found, the algorithm terminates and the inconsistencies in μ is fixed FIXMATCHING kernel which implements: $\mu(v) \leftarrow -1$ for any v with $\mu(\mu(v)) \neq v$. These inconsistencies do not prevent the algorithm obtaining a maximum cardinality matching. When all GPU kernels are done, such inconsistencies can only exist at the column entries of μ and the row matching will be correct. In fact, if there exist a perfect matching in G all the entries in μ are correct and no fix operation is required.

Algorithm 6 G-PR-KRNL

Input: $G = (V_R \cup V_C, E)$, μ, ψ
1: $actExists \leftarrow$ **false**
2: **for all** column vertices $v \in V_C$ **in parallel do**
3: **if** $\mu(u) = -1$ **or** $\mu(\mu(v)) \neq v$ **then**
4: $actExists \leftarrow$ **true**
5: $\psi_{min} \leftarrow m + n$
6: $u \leftarrow -1$
7: **for all** $u' \in \Gamma(v)$ **do**
8: **if** $\psi(u') < \psi_{min}$ **then**
9: $\psi_{min} \leftarrow \psi(u')$
10: $u \leftarrow u'$
11: **if** $\psi_{min} = \psi(v) - 1$ **then**
12: **break**
13: **if** $\psi_{min} < m + n$ **then**
14: $\mu(u) \leftarrow v$
15: $\mu(v) \leftarrow u$
16: $\psi(v) \leftarrow \psi_{min} + 1$
17: $\psi(u) \leftarrow \psi_{min} + 2$
18: **else**
19: $\mu(v) \leftarrow -2$

C. Reducing the number of GPU threads

In the G-PR-KRNL kernel, the proposed parallel push-relabel-based algorithm uses n GPU threads to traverse the active column vertices. However, thanks to the initial matching heuristic, the number of unmatched columns is much smaller than n. Furthermore, we observed that most of the G-PR-KRNL executions are performed only with a few number of active columns, since it is harder to find augmenting paths when they are longer. This is usually the case when the deficiency gets smaller, which happens towards the end of the execution. In order to reduce the number of threads, we propose two main modifications on the algorithm.

1) Keeping the list of active columns: The first modification is to keep a list of active columns and have the threads work on those active columns (instead of all columns). Assume that we have an array $\mathcal{A}$. Let $v = \mathcal{A}(i)$ be an active column and $v' \in V_C$ be the column which becomes active once v is consumed. Since there can be at most one such column, in a sequential execution, setting $\mathcal{A}(i) \leftarrow v'$ is sufficient to keep the id of the new active columns produced during the push-relabel operations. When no new active column is produced after a consumption, one can set the corresponding value to -1. However, in a concurrent execution, some of the push-relabel operations may need to be rolled back due to the conflicts.

As explained above, we need to keep the id of each consumed column to roll it back in case of conflicts. Hence, a single array is not sufficient to support correct and concurrent execution of push operations. In our implementation, we used two arrays $\mathcal{A}_c$ and $\mathcal{A}_p$ to maintain such a list and support concurrent operations on it at the same time. Initially, both arrays contain the unmatched column indices. In the course of the algorithm, the entries will contain either the ids of the active columns or -1 which appear when an augmentation (only a single push) takes place without producing new

active columns.

Algorithm 7 G-PR: GPU-based Push-Relabel Matching

Input: $G = (V_R \cup V_C, E)$, μ
1: $loop \leftarrow 0$
2: $actExists \leftarrow$ **true**
3: $shrink \leftarrow$ **false**
4: $iterGR \leftarrow 0$
5: **while** $actExists$ **do**
6: **if** $loop = iterGR$ **then**
7: $\langle maxLevel, \psi \rangle \leftarrow$ G-GR(G, μ)
8: $iterGR \leftarrow$ GETITERGR$(maxLevel, loop, iterGR)$
9: $shrink \leftarrow$ **true**
10: $actExists \leftarrow$ **false**
11: **if** $shrink$ **and** $|\mathcal{A}_c| \geq 512$ **then**
12: $\langle actExists, \mathcal{A}_c, iA \rangle \leftarrow$ G-PR-SHRKRNL$(G, \mu, \psi, loop, \mathcal{A}_c, \mathcal{A}_p, iA)$
13: $shrink \leftarrow$ **false**
14: **else**
15: $\langle actExists, \mathcal{A}_c, iA \rangle \leftarrow$ G-PR-INITKRNL$(G, \mu, \psi, loop, \mathcal{A}_c, \mathcal{A}_p, iA)$
16: **if** $actExists$ **then**
17: $\langle \mu, \psi, \mathcal{A}_c, \mathcal{A}_p \rangle \leftarrow$ G-PR-PUSHKRNL$(G, \mu, \psi, loop, \mathcal{A}_c, \mathcal{A}_p, iA)$
18: Swap $\mathcal{A}_c$ and $\mathcal{A}_p$
19: $loop \leftarrow loop + 1$
20: $\mu \leftarrow$ FIXMATCHING(μ)
21: **return** μ

In a single loop of the new G-PR, given in Algorithm **??**, push-relabel operations are performed in two steps. In the first step, the algorithm fixes $\mathcal{A}_c$ by using $\mathcal{A}_p$ from the previous iteration with the kernel G-PR-INITKRNL given in Algorithm **??**. The fix operation is done in two stages: first the conflicting operations in the previous execution are detected (line **??**) and then they are rolled back (line **??**). In addition to $\mathcal{A}_c$ and $\mathcal{A}_p$, we used an additional array iA of size n where $iA(v) = loop$ if and only if v is an active column. While fixing $\mathcal{A}_c$, G-PR-INITKRNL also sets the corresponding entries to $loop$ (line **??**). This array will be used to avoid duplicate processing of an active column by two different threads while performing push-relabel operations. Note that the number of GPU threads for this kernel execution is $|\mathcal{A}_p|$ which is expected to be much smaller than n after the greedy matching heuristic.

Algorithm 8 G-PR-INITKRNL

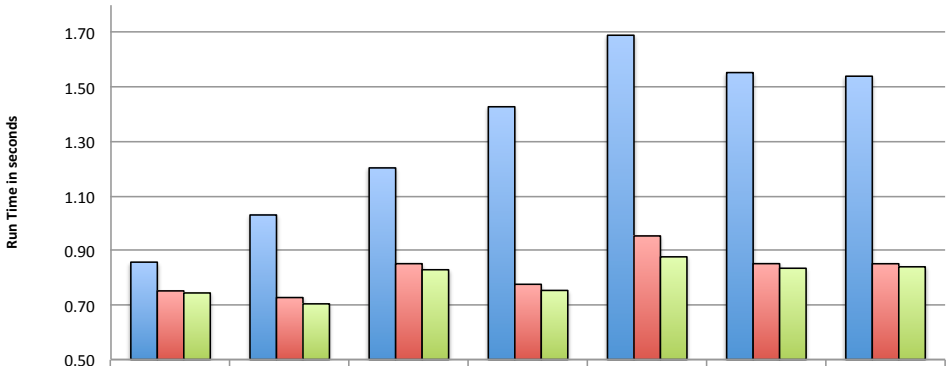
Input: $G = (V_C \cup V_R, E), \mu, \psi, loop, \mathcal{A}_c, \mathcal{A}_p, iA$
1: $actExists \leftarrow$ **false**
2: **for all** $i \in \{1, \ldots, |\mathcal{A}_p|\}$ **in parallel do**
3: $v \leftarrow \mathcal{A}_p(i)$
4: **if** $v \neq -1$ **then**
5: **if** $\mu(v) = -1$ **or** $v \neq \mu(\mu(v))$ **then**
6: $\mathcal{A}_c(i) \leftarrow \mathcal{A}_p(i)$
7: **else**
8: $v \leftarrow \mathcal{A}_c(i)$
9: **if** $v \neq -1$ **then**
10: $iA(v) \leftarrow loop$
11: $actExists \leftarrow$ **true**

Once $\mathcal{A}_c$ is fixed, the push-relabel operations are performed by G-PR-PUSHKRNL given in Algorithm **??**. The kernel works along the same lines with the previous GPU-based concurrent push-relabel kernel G-PR-KRNL. However, it uses less threads. The main difference is keeping the ids of the active columns produced throughout the execution (line 18) and the extra check $iA(\mu(u)) \stackrel{?}{\neq} loop$ at line 13. This check is necessary due to concurrency; an already active column v which is being processed can be inserted to $\mathcal{A}_p$ in the same kernel execution. Although, this does not create a problem for the current execution, if G-PR-INITKRNL roll backs the operation on v, in the next execution v will be processed by two different threads. To avoid this problem, a column, which is already active at the beginning of the iteration is forbidden in $\mathcal{A}_p$. After completing G-PR-PUSHKRNL, at the end of each iteration, we swap the current and previous active column arrays for the next iteration (line 18 of Algorithm **??**).

Algorithm 9 G-PR-PUSHKRNL

Input: $G = (V_R \cup V_C, E), \mu, \psi, loop, \mathcal{A}_c, \mathcal{A}_p, iA$
1: **for all** $i \in \{1, \ldots, |\mathcal{A}_p|\}$ **in parallel do**
2: $v \leftarrow \mathcal{A}_c(i)$
3: **if** $v \neq -1$ **then**
4: $\psi_{min} \leftarrow m + n$
5: $u \leftarrow -1$
6: **for all** $u' \in \Gamma(v)$ **do**
7: **if** $\psi(u') < \psi_{min}$ **then**
8: $\psi_{min} \leftarrow \psi(u')$
9: $u \leftarrow u'$
10: **if** $\psi_{min} = \psi(v) - 1$ **then**
11: **break**
12: **if** $\psi_{min} < m + n$ **then**
13: **if** $\mu(u) = -1$ or $iA(\mu(u)) \neq loop$ **then**
14: $\mu(u) \leftarrow v$
15: $\mu(v) \leftarrow u$
16: $\psi(v) \leftarrow \psi_{min} + 1$
17: $\psi(u) \leftarrow \psi_{min} + 2$
18: $\mathcal{A}_p(i) \leftarrow w$
19: **else**
20: $\mu(v) \leftarrow -2$
21: $\mathcal{A}_c(i) \leftarrow -1$
22: $\mathcal{A}_p(i) \leftarrow -1$
23: **else**
24: $\mathcal{A}_p(i) \leftarrow -1$

2) Dynamic compression of active-column lists: The size of the active column arrays used in Algorithms **??** and **??** is equal to the number of unmatched columns at the beginning. When the deficiency is smaller, we observed that the main loop of the algorithm iterates more in order to increase the matching size. Hence, the impact of a reduction on the size of the active column arrays will be significant. Thus, as a second improvement, we periodically shrink these arrays after each global-relabel operation by using G-PR-SHRKRNL (line 12). The kernel works along the same lines with G-PR-INITKRNL. But instead of directly writing the new active columns to $\mathcal{A}_c$, an initial pass on the previous and the current active columns is performed in order to obtain the number of local active columns within each thread. Then the threads perform a prefix sum operation on these counts. After the prefix sum, each



	adaptive, 0.3	adaptive, 0.7	adaptive, 1	adaptive, 1.5	adaptive, 2	fix, 10	fix, 50
G-PR-First	0.86	1.03	1.20	1.43	1.69	1.55	1.54
G-PR-NoShr	0.75	0.73	0.85	0.78	0.95	0.85	0.85
G-PR-Shr	0.74	0.70	0.83	0.75	0.88	0.83	0.84

Fig. 1. The comparison of the proposed G-PR algorithms with different global-relabeling strategies.

thread has a private write region on $\mathcal{A}_c$. They traverse the previous active column array again, and instead of counting, this time they write the ids to their private regions. Hence, after G-PR-SHRKRNL the size of $\mathcal{A}_c$ and $\mathcal{A}_p$ is reduced to the exact number of active columns. We observed that although the shrinking can yield significant improvements, after some number of active columns the improvement does not compensate its overhead. Hence, we decide to use the shrink operation only when $|\mathcal{A}_c| \geq 512$ (line 11 of Algorithm **??**).

IV. EXPERIMENTS

The running time of the proposed implementations are compared against the sequential PR implementation (PR) [**?**], the multicore parallel implementations P-DBFS [**?**], and a GPU implementation of HKDW [**?**]. For the sequential PR implementation, we tried a set of k values to set the frequency of global-relabeling to $k \times (m + n)$. For our data set $k = 0.5$ was slightly better. Hence, we used it in our experiments. The CPU implementations are tested on a computer with 2.27GHz dual quad-core Intel Xeon CPUs with 2-way hyper-threading and 48GB main memory. The algorithms are implemented in C++ and OpenMP. The GPU implementations are tested on NVIDIA Tesla C2050 with usable 2.6GB of global memory. Tesla C2050 is equipped with 14 multiprocessors each containing 32 CUDA cores, totaling 448 CUDA cores. The implementations are compiled with gcc-4.4.4, cuda-4.2.9 and -O2 optimization flag. For the multicore algorithms, 8 threads are used. A standard heuristic called the cheap matching (see [**?**], [**?**]) is used to initialize all tested algorithms. We compare the running time of the matching algorithms after this common initialization.

The algorithms are run on bipartite graphs corresponding to 28 matrices from variety of classes at UFL matrix collection [**?**]. The graphs are chosen from a large dataset given in [**?**]. The graphs for which all of the sequential algorithms, Pothen-Fan-Plus [**?**], Hopcroft-Karp [**?**], and the push-relabel algorithm (PR [**?**]), take less than one second are filtered out.

We first compare the performance of the proposed GPU algorithm with different parameters. Figure **??** shows the geometric means of the runtimes on different sets. In the figure, G-PR-First corresponds to our first implementation explained in Algorithm **??**. G-PR-Shr corresponds to the implementation presented in Algorithm **??**, while G-PR-NoShr is the one that omits the shrinking mechanism (G-PR-SHRKRNL functions omitted). Different strategies are used to specify the frequency of the global relabeling. Here, (**adaptive, k**) refers to the the adaptive global-relabeling strategy explained in Section **??**. That is, the next global relabeling is performed after $k \times maxLevel$ push-relabel kernel calls. For the (**fix, k**) strategy, the next global-relabeling operation is performed after k push-relabel kernel executions. As the figure shows, we obtain the best results with (**adaptive, 0.3**) and (**adaptive, 0.7**). Having a fix frequency for all graphs throughout the execution is outperformed by the adaptive heuristic for almost all configurations. The results verify the necessity of adjusting the frequency of the global relabeling with respect to the graph structure and the current matching. The figure also shows that the proposed G-PR-active algorithm improves the performance of each configuration by 14% to 84%, as it decreased the divergence of the GPU threads. Shrinking the list of active columns throughout the execution improves the performance of the algorithms by another 2–8%. Since G-PR-shrink obtains the best performance with (**adaptive, 0.7**) configuration, we only compare the performance of this configuration with other implementations in the literature.

Figure **??** shows the speedup profiles of G-PR, G-HKDW, and P-DBFS on the experiment set. The speedups are calculated with respect to the sequential PR algorithm. A point (x,y) in the plots corresponds to the probability y of obtaining at least x speedup. As the plot shows, G-PR has a better speedup profile: with 39% probability it obtains a speedup at least 5. This probability is 21% and 14% for G-HKDW and P-DBFS, respectively. Furthermore, the proposed algorithm is faster than the sequential PR algorithm for 82% of the graphs.

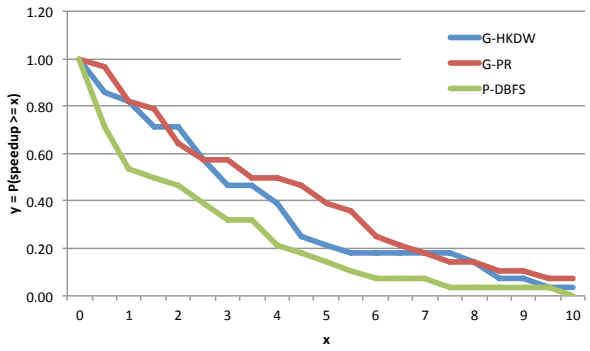


Fig. 2. Speedup profiles of the parallel algorithms. A point (x,y) in the plots corresponds to the probability of obtaining at least x speedup is y.

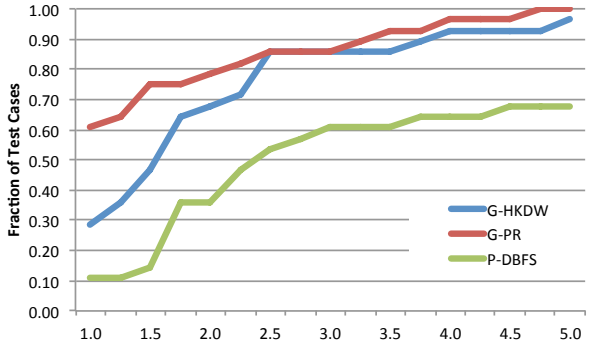


Fig. 3. Performance profiles of the parallel algorithms. A point (x,y) in this plot means that with y probability, the algorithm obtains a performance that is at most x times worse than the best running time.

Although the performance of G-HKDW is close to that of G-PR, both G-HKDW and P-DBFS are outperformed by the proposed G-PR algorithm. On the average, the proposed algorithm is 1.30 and 2.82 times faster than G-HKDW and P-DBFS, respectively, when geometric means of the runtimes are compared.

Figure **??** shows the performance profiles of multicore and manycore algorithms. A point (x,y) in this plot means that with y probability, the algorithm obtains a performance that is at most x times worse than the best running time. The plot clearly shows the separation among G-PR and the other algorithms, thus marking G-PR as the fastest in most cases. For 75% of the cases, G-PR is at most 1.5 times worse than the best algorithm. These percentages are 46% and 14% for G-HKDW and P-DBFS, respectively. In particular, G-PR obtains the best performance in 61% of the selected graphs.

Figure **??** shows the indivual speedups of G-PR on the each graph. The x axis gives the graph ids given in Table **??** and the graphs are ordered with the increasing number of rows. The proposed algorithm obtains a speedup on 23 (out of 28) of the graphs. The maximum speedup achieved is on delaunay_n24 as 12.60, while the minimum speedup is obtained as 0.31 on hugetrace-00000 graph. The overall average speedup is 3.05. Table **??** gives the actual running times of the best GPU and multicore algorithms, together with the sequential PR algorithm. As seen from this table, on 15 of the graphs, the GPU algorithm is faster than the other algorithms.

V. Concluding Remarks

We proposed a parallel push-relabel-based maximum cardinality matching algorithm for bipartite graphs on GPUs. We presented a lock- and atomic-free implementation and propose a set of modifications to obtain a better performance. We investigated the effect of the global-relabeling frequency and developed a strategy to amortize the cost of global and push relabeling. We presented experiments on various graphs and compared the performance of the proposed implementation against sequential, multicore, and manycore algorithms from the literature. The experiments showed that the proposed GPU implementation is faster than the existing implementations. We obtained speedups varied from 0.31 to 12.60, averaging 3.05, on a set of 28 graphs with respect to a recent sequential push-relabel-based implementation.

As a future work, we intend to investigate more architectural and algorithmic advancements to improve the performance. One idea we want to study is the concurrent execution of global-relabeling and push-relabel kernels [**?**]. As far as we know, there is no GPU implementations of these techniques. After the Fermi architecture, NVIDIA GPUs support concurrent kernel executions with streams. Hence, it may be promising to occupy the device with two kernels when there is not enough parallelism with a single kernel.

Acknowledgment

This work was supported in parts by the NSF grants OCI-0904809 and OCI-0904802.

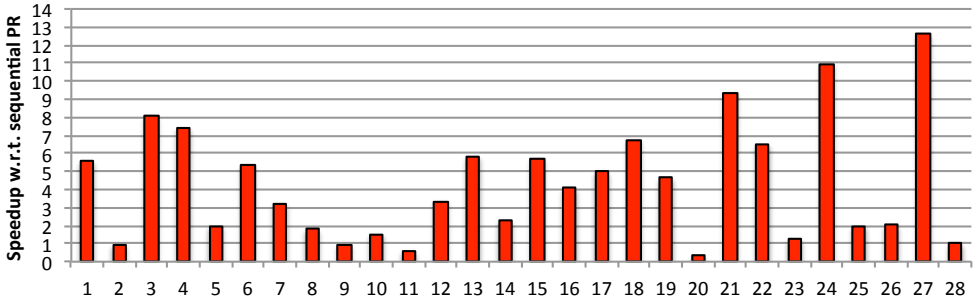


Fig. 4. Individual speedups of G-PR algorithm on each graph instance.

TABLE I

ACTUAL RUNNING TIME OF EACH ALGORITHM FOR ALL GRAPHS TOGETHER WITH THE NUMBER OF ROWS, COLUMNS, AND EDGES IN EACH GRAPH. IM AND MM GIVES THE CARDINALITY OF THE INITIAL AND MAXIMUM MATCHING FOR EACH GRAPH, RESPECTIVELY. THE GEOMETRIC MEANS OF THE RUN TIMES OF G-PR, G-HKDW, P-DBFS, AND PR ARE GIVEN IN THE BOTTOM ROW.

ID	Graph	#Row	#Cols	#Edges	IM	MM	G-PR	G-HKDW	P-DBFS	PR
1	amazon0505	410,236	410,236	3,356,824	332,972	395,397	**0.09**	0.18	22.70	0.52
2	coPapersDBLP	540,486	540,486	15,245,729	510,992	540,226	0.62	**0.42**	6.27	0.59
3	amazon-2008	735,323	735,323	5,158,388	587,877	641,379	0.12	**0.11**	0.18	0.93
4	flickr	820,878	820,878	9,837,214	285,241	367,147	**0.13**	0.22	0.35	0.99
5	eu-2005	862,664	862,664	19,235,140	642,027	652,328	**0.40**	1.54	0.94	0.80
6	delaunay_n20	1,048,576	1,048,576	3,145,686	993,174	1,048,576	0.06	**0.04**	0.09	0.32
7	kron_g500-logn20	1,048,576	1,048,576	44,620,272	431,854	513,334	**0.38**	0.60	8.19	1.24
8	roadNet-PA	1,090,920	1,090,920	1,541,898	916,444	1,059,398	0.33	**0.14**	0.29	0.59
9	in-2004	1,382,908	1,382,908	16,917,053	781,063	804,245	0.58	1.44	2.16	**0.56**
10	roadNet-TX	1,393,383	1,393,383	1,921,660	1,158,420	1,342,440	0.45	**0.14**	0.33	0.69
11	Hamrle3	1,447,360	1,447,360	5,514,242	1,211,049	1,447,360	0.94	1.36	2.70	**0.56**
12	as-Skitter	1,696,415	1,696,415	11,095,298	891,280	1,035,521	**0.34**	0.49	1.89	1.13
13	GL7d19	1,911,130	1,955,309	37,322,725	1,904,144	1,911,130	**0.24**	0.58	0.38	1.38
14	roadNet-CA	1,971,281	1,971,281	2,766,607	1,668,268	1,913,589	0.68	**0.34**	0.53	1.55
15	delaunay_n21	2,097,152	2,097,152	6,291,408	1,987,326	2,097,152	0.18	**0.13**	0.21	1.06
16	kron_g500-logn21	2,097,152	2,097,152	91,042,010	812,883	964,679	**0.68**	0.99	1.50	2.77
17	wikipedia-20070206	3,566,907	3,566,907	45,030,389	1,623,931	1,992,408	**0.62**	1.09	5.24	3.11
18	patents	3,774,768	3,774,768	14,970,767	1,892,820	2,011,083	**0.54**	0.88	0.84	3.65
19	com-livejournal	3,997,962	3,997,962	34,681,189	2,577,642	3,608,272	**2.08**	4.58	22.46	9.67
20	hugetrace-00000	4,588,484	4,588,484	6,879,133	4,581,148	4,588,484	2.71	1.96	**0.83**	0.84
21	soc-LiveJournal1	4,847,571	4,847,571	68,993,773	2,831,783	3,835,002	**1.35**	3.32	14.35	12.66
22	ljournal-2008	5,363,260	5,363,260	79,023,142	3,941,073	4,355,699	**1.54**	2.37	10.30	10.01
23	italy_osm	6,686,493	6,686,493	7,013,978	6,438,492	6,644,390	5.46	5.86	**1.20**	6.84
24	delaunay_n23	8,388,608	8,388,608	25,165,784	7,950,070	8,388,608	**0.81**	0.96	1.26	8.86
25	wb-edu	9,845,725	9,845,725	57,156,537	4,810,825	5,000,334	**2.00**	33.82	8.61	3.94
26	hugetrace-00020	16,002,413	16,002,413	23,998,813	15,535,760	16,002,413	14.19	**7.90**	393.13	28.69
27	delaunay_n24	16,777,216	16,777,216	50,331,601	15,892,194	16,777,216	**1.83**	1.98	2.41	23.01
28	hugebubbles-00000	18,318,143	18,318,143	27,470,081	18,303,614	18,318,143	13.65	13.16	**3.55**	13.51
						GEOMEAN	**0.70**	0.92	1.99	2.15