GPU accelerated maximum cardinality matching algorithms for bipartite graphs

Bora Uçar

CNRS and LIP, ENS Lyon, France

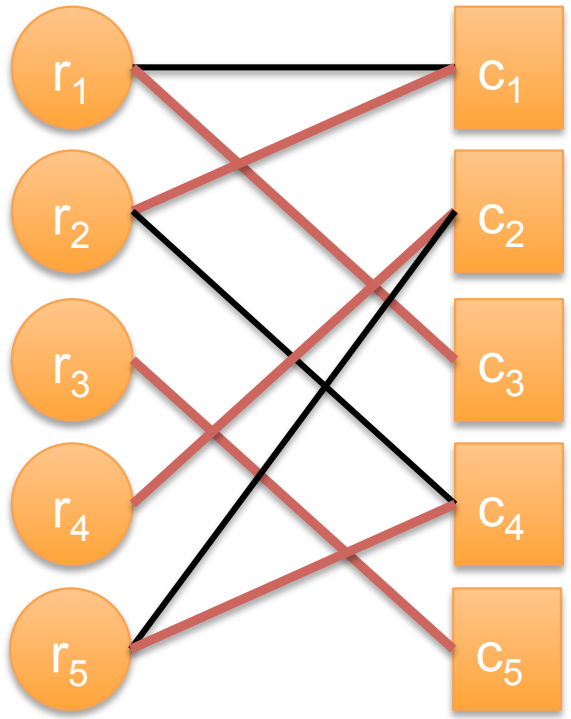
Europar 2013, 26–30 Auguest, 2013, Aachen, Germany

Joint work with:

Mehmet Deveci
Ümit V. Çatalyürek Kamer Kaya
BMI (and ECE for MD & UVÇ),
The Ohio State University

Bipartite graphs and matchings

- $G = (R \cup C, E)$ is a bipartite graph with the vertex set $R \cup C$ where $R \cap C = \emptyset$, and all edges contain one vertex in R other in C.
- A matching $\mathcal{M}$ in a graph G is a subset of edges E where a vertex in $R \cup C$ is in at most one edge in $\mathcal{M}$.
- Perfect matching all vertices in R or C are matched, e.g., $(r_1, c_3), (r_2, c_1), (r_3, c_5), (r_4, c_2), (r_5, c_4)$.

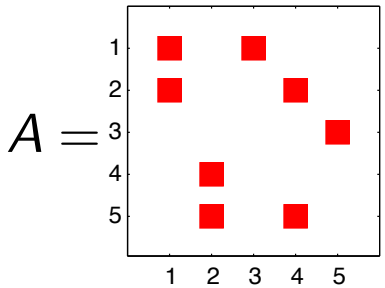


Problem: Find a matching of maximum cardinality.

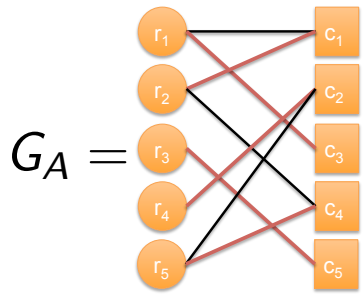
Outline

Matrices, bipartite graphs and matchings

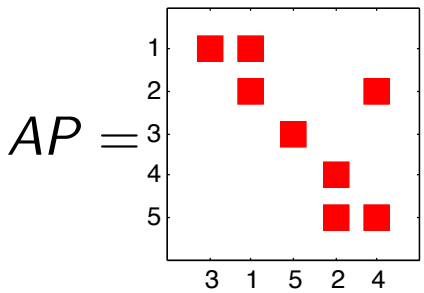
Motivation: Given an $n \times n$ sparse matrix A, find a permutation of the columns so that the diagonal of the permuted matrix is zero free.



1. Take the associated bipartite graph $G_A = (R \cup C, E)$
   - R corresponds to the set of rows, C to the set of columns
   - $(r_i, c_j) \in E$ iff $a_{ij} \neq 0$.



2. Compute a perfect matching in G_A.
3. Permute columns according to the matching.

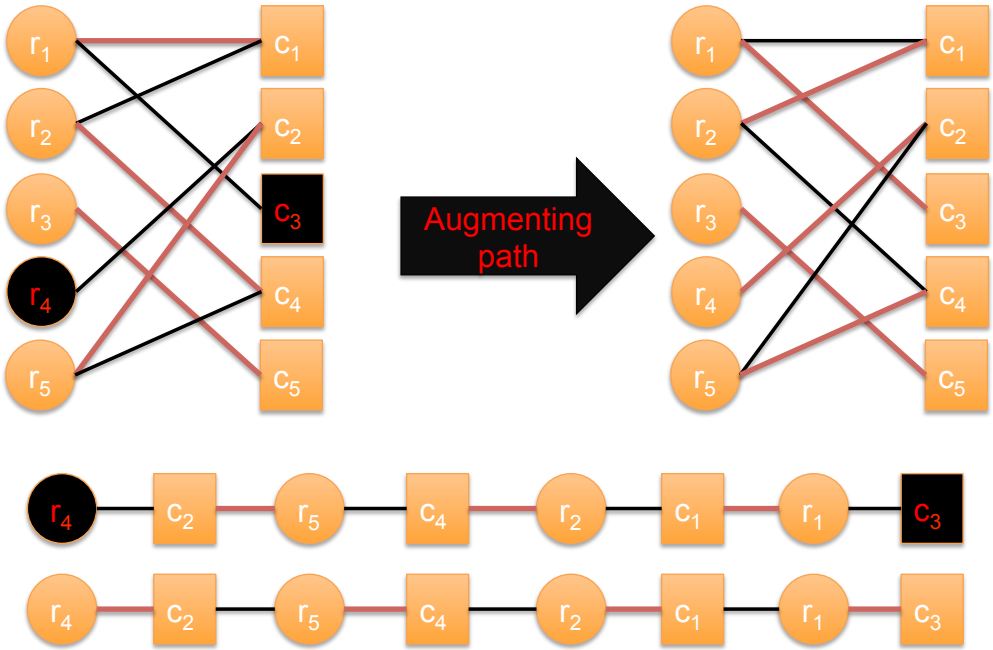


The permuted form can be used to detect reducibility of A; if so substantial savings are possible while solving the associated linear system.

Augmenting paths

Alternating path: A path in G is $\mathcal{M}$-alternating if its edges are alternatively in $\mathcal{M}$ and not in $\mathcal{M}$.

Augmenting path: An $\mathcal{M}$-alternating path $\mathcal{P}$ is called $\mathcal{M}$-augmenting if the start and end vertices of $\mathcal{P}$ are both unmatched.



All (exact, deterministic) algorithms are based on augmenting paths: start with possible empty matching and augment (theorem of Berge).

Algorithms for bipartite mathching

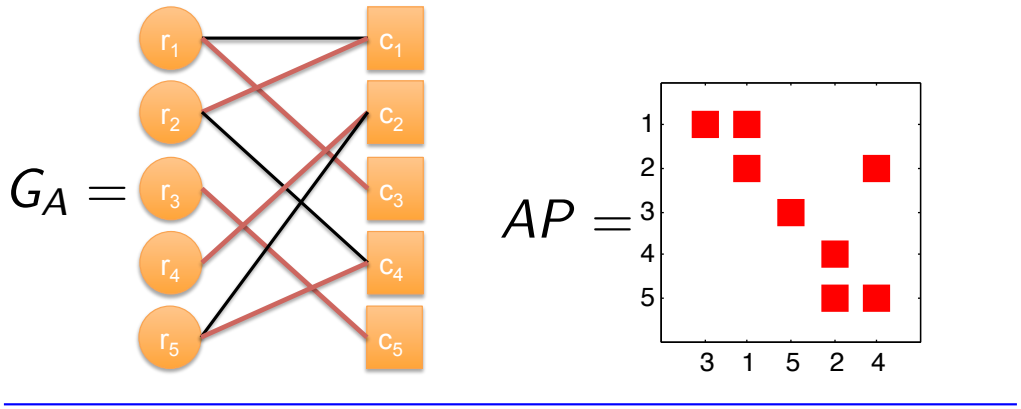
Alg.	Description	Complexity
DFSB	DFS. Forms the basis of many algorithms.	$\mathcal{O}(n\tau)$
BFSB	BFS. Quite common (the algorithm FF in [Melhorn and Näher,'99]).	$\mathcal{O}(n\tau)$
MC21A	DFS+Lookahead [Duff,'81] and dmperm in Matlab [Davis,'06]—the most wide-spread?	$\mathcal{O}(n\tau)$
PF	Phases of disjoint DFSs [Pothen and Fan,'90].	$\mathcal{O}(n\tau)$
HK	Shortest disjoint augmenting paths [Hopcroft and Karp,'73].	$\mathcal{O}(\sqrt{n}\tau)$
HKDW	HK+Disjoint DFS [Duff and Wiberg,'88].	$\mathcal{O}(\sqrt{n}\tau)$
ABMP	Combined DFS and BFS [Alt, Blum, Mehlhorn, and Paul,'91].	$\min\{\mathcal{O}(\sqrt{n}\tau)$ $\mathcal{O}(n^{1.5}\sqrt{\tau/\log n})\}$
PF+	A simple modification of PF [Duff, Kaya, and U.,'10].	$\mathcal{O}(n\tau)$
PR	Push-relabel [Cherkassky, Goldberg, Martin, Setubal, Stolfi,'98]; Bounds on distances to free vertices.	$\mathcal{O}(\sqrt{n}\tau)$
PseudoFlow	Prefixes and suffixes of augmenting paths [Hochbaum'98 and Chandran and Hochbaum'11]	$\mathcal{O}(n\tau)$

Some recent parallelization studies

- Undirected graph
  - weighted, unweighted, approximate, GPU, MPI, external memory: Brin, Osipov, Sanders, Schulz, Sitchinava, Session F2, (EuroPar'13).
  - weighted, unweighted, heuristic, GPU: Fagginger Auer and Bisseling'12.
  - weighted, GPU, multicore: Halappanavar, Feo, Villa, Tumeo, and Pothen'12.
  - weighted, greedy, multicore: Çatalyürek, Deveci, Kaya, U.'12.
- Bipartite graph
  - weighted, GPU: Vasconcelos and Rosenhahn'09.
  - unweighted, multicor: Azad, Halappanavar, Rajamanickam, Boman, Khan, Pothen'12.

We propose: bipartite unweighted, GPU.

Outline



Proposed algorithms

- Based on HK and HKDW: Use BFS to locate a set of shortest augmenting paths, augment along a maximal set of them using DFS.
- HKDW adds one more DFS step to augment along the remaining paths (not shortest).
- Keep the BFS part; the DFS part does not propose efficiency.

Overall description

- HK: Find a set of shortest augmenting paths, alternate along all of them (some of them will be realized)
- HKDW: Find the set of augmenting paths, alternate along all of them (some of them will be realized).
- The worst case running time complexity increases, $\mathcal{O}(n\tau)$ instead of $\mathcal{O}(\sqrt{n}\tau)$. We trade that to achieve fine-grained parallelism.

Proposed algorithms: Main one, similar to HKDW

Algorithm 1: ALL AUGMENTING PATHS (APFB)

```
   Data: cxadj, cadj, nc, nr, rmatch, cmatch
 1 augmenting_path_found ← true;
 2 while augmenting_path_found do
 3     bfs_level ← L0;
 4     InitBfsArray(bfs_array, cmatch, L0);
 5     vertex_inserted ← true;
 6     while vertex_inserted do
 7         predecessor ←Bfs(bfs_level, bfs_array, cxadj, cadj, nc, rmatch,
 8                          vertex_inserted, augmenting_path_found);
 9
10
11         bfs_level ← bfs_level + 1;
12     ⟨cmatch, rmatch⟩ ← Alternate (cmatch, rmatch, nc, predecessor);
13     ⟨cmatch, rmatch⟩ ← FixMatching (cmatch, rmatch);
```

BFS uses alternating paths, starts from unmatched columns, tries to reach unmatched rows.

Needed to avoid atomic operations and locks.

Proposed algorithms: BFS kernel function 1

The visited column vertex in the current level

```
4   if bfs_array[col_vertex] = bfs_level then
5       for j from cxadj[col_vertex] to cxadj[col_vertex + 1] do
6           neighbor_row ← cadj[j];
7           col_match ← rmatch[neighbor_row];
8           if col_match > −1 then
9               if bfs_array[col_match] = L0 −1 then
0                   vertex_inserted ← true;
1                   bfs_array[col_match] ← bfs_level + 1;
2                   predeccesor[neighbor_row] ← col_vertex;
3           else
4               if col_match=−1 then
5                   rmatch[neighbor_row] ← −2;
6                   predeccesor[neighbor_row] ← col_vertex;
7                   augmenting_path_found ← true;
```

Unvisited (matched) column vertex is found

Unmatched row vertex is found

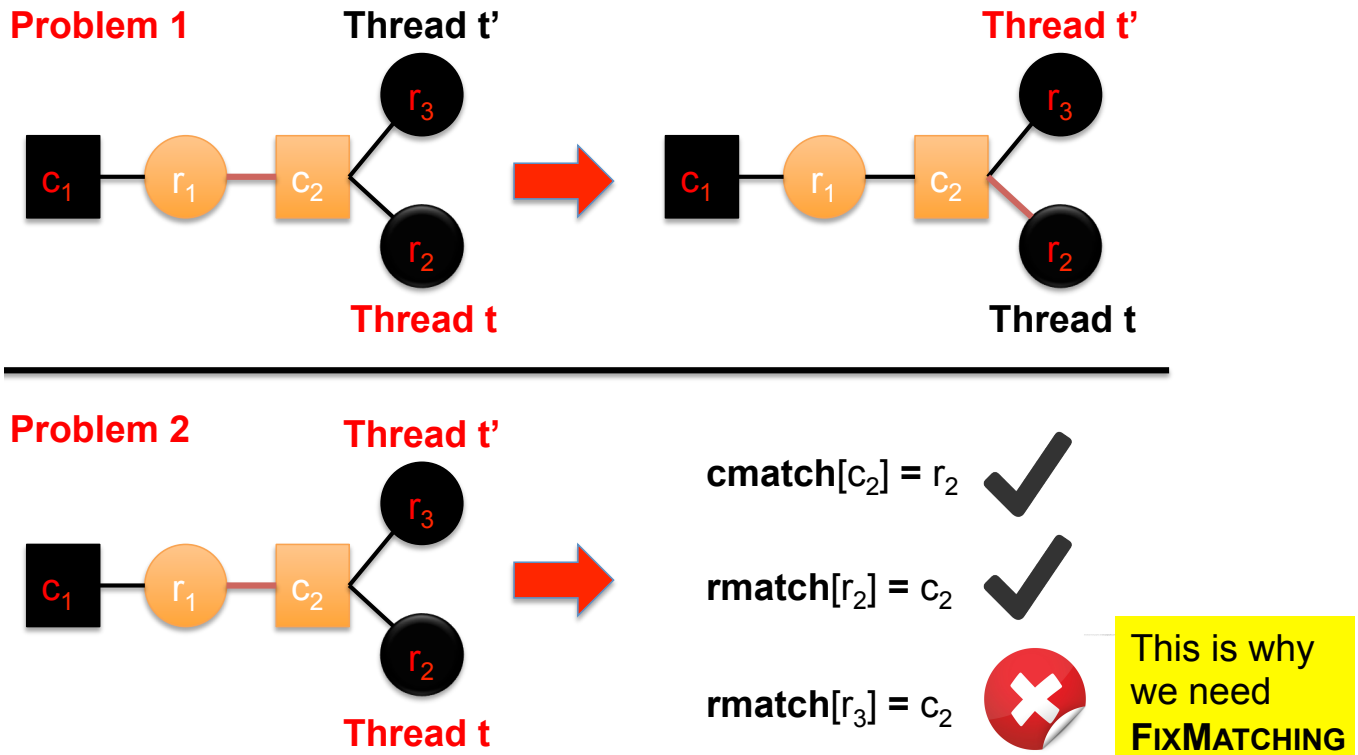
Proposed algorithms: Alternate

Algorithm 3: ALTERNATE

```
   Data: cmatch, rmatch, nc, nr, predecessor
 1 process_vcnt ← getProcessCount(nr);
 2 for i from 0 to process_vcnt − 1 do
 3     row_vertex ← i × tot_thread_num + tid;
 4     if rmatch[row_vertex] = −2 then
 5         while row_vertex ≠ −1 do
 6             matched_col ← predecessor[row_vertex];
 7             matched_row ← cmatch[matched_col] ;
 8             if predecessor[matched_row] = matched_col then
 9                 break;
10             cmatch[matched_col] ← row_vertex;
11             rmatch[row_vertex] ← matched_col;
12             row_vertex ← matched_row;
```

Line 3 $\rightsquigarrow$ coalesced access to the memory.

Proposed algorithms: Fix Matching



FIXMATCHING:

$rmatch[r] \leftarrow -1$ for any r satisfying $cmatch[rmatch[r]] \neq r$

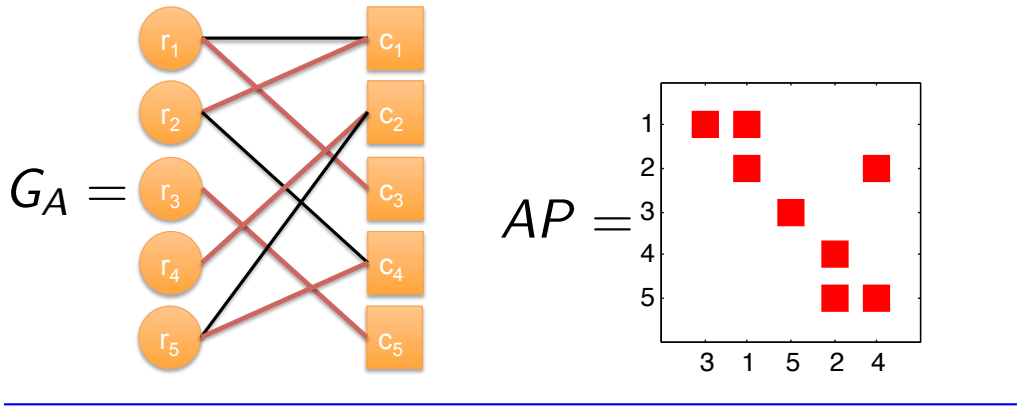
Proposed algorithms: BFS kernel modified

...so that:

early exits; if an augmenting path found for a column, no more BFS's continue for the same column.

helps Alternate: mark the start and the end of the augmenting paths (so that alternate works along correct augmenting paths).

Outline



Experiments

- The sequential HK and PFP implementations Duff, Kaya, and U'11.
- Multicore implementations P-PFP, P-DBFS, and P-HK from Azad et al.'12 on 8 threads.

CPU: 2.27GHz dual quad-core Intel Xeon CPUs with 2-way hyper-threading and 48GB main memory (C++ and OpenMP).

GPU: NVIDIA Tesla C2050 with usable 2.6GB of global memory (14 multiprocessors each containing 32 CUDA cores).

- gcc-4.4.4, cuda-4.2.9 and -O2 optimization flag.
- A standard heuristic is used to initialize all algorithms.

The execution times of the GPU algorithms exclude memory copy time (when included decreases the reported mean speedups across all data set by at most 6%.)

Experiments: Data set and GPU algorithms

Data

- 70 large matrices from UFL collection. "O" original set, "RCP" random row/column permutations.
- report on those matrices for which one of the sequential algorithms took more than one second (O_S1, 28 matrices; and RCP_S1, 50 matrices).
- O_Hardest20 and RCP_Hardest20 that contain the set of 20 matrices on which the sequential algorithms required the longest runtime.

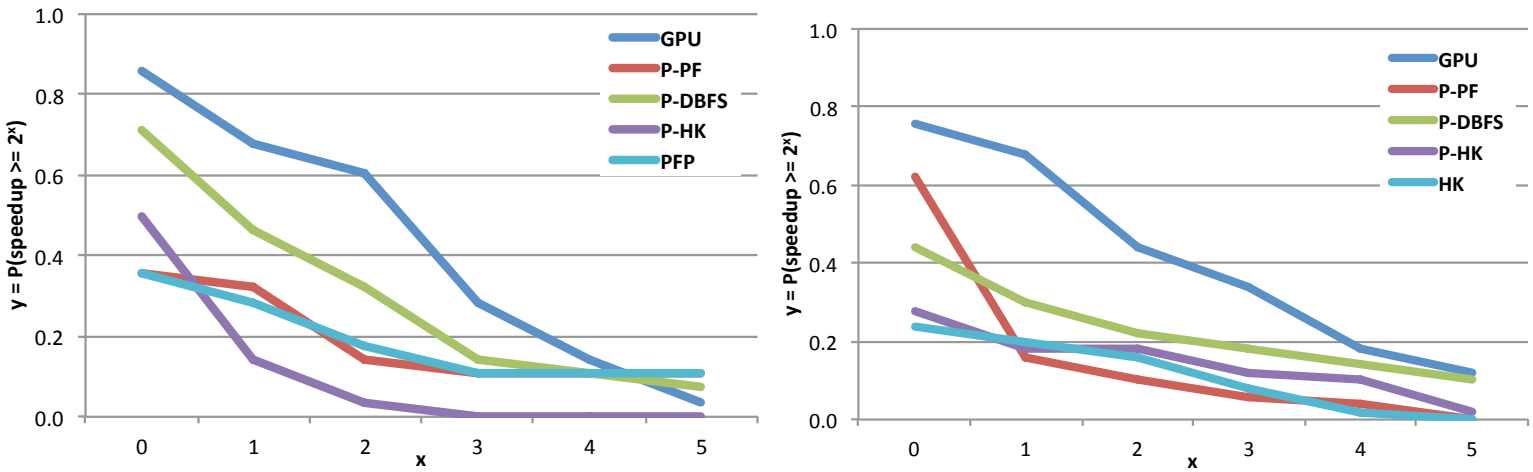
GPU algorithms

Geometric mean of the runtime (in seconds) on different sets of instances

	APFB				APsB			
	GPUBFS		GPUBFS-WR		GPUBFS		GPUBFS-WR	
	MT	CT	MT	CT	MT	CT	MT	CT
O_S1	2.96	1.89	2.12	**1.34**	3.68	2.88	2.98	2.27
O_Hardest20	4.28	2.70	3.21	**1.93**	5.23	4.14	4.20	3.13
RCP_S1	3.66	3.24	1.13	**1.05**	3.52	3.33	2.22	2.14
RCP_Hardest20	7.27	5.79	3.37	**2.85**	12.06	10.75	8.17	7.41

Log-scaled speedup profiles

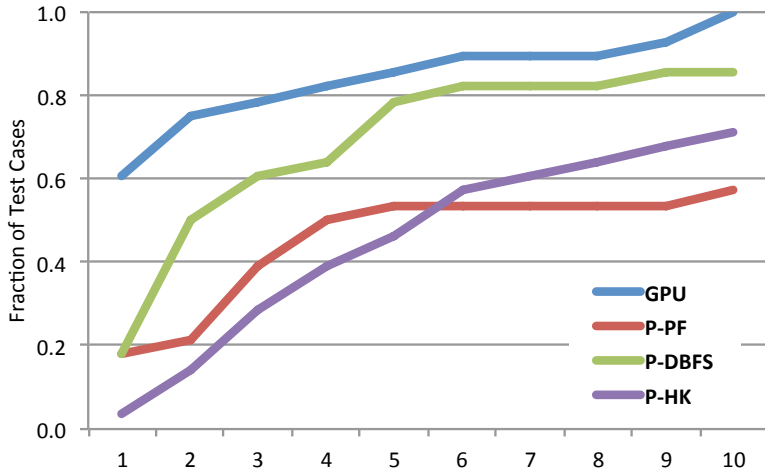
Best identified GPU algorithm and the multicore ones:



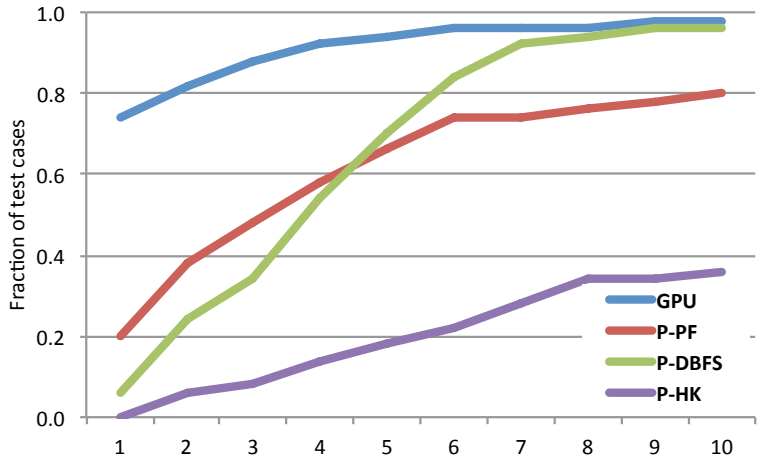
(a) Original graphs (b) Permuted graphs

- (x, y): the probability of obtaining at least 2^x speedup is y.
- The speedups wrt the fastest of the seq. algorithms (min(PFP and HK)).
- GPU algorithm has the best overall speedup (faster than HK in 86% of the original graphs, while it is faster than PFP on 76% on the permuted graphs).

Performance profiles



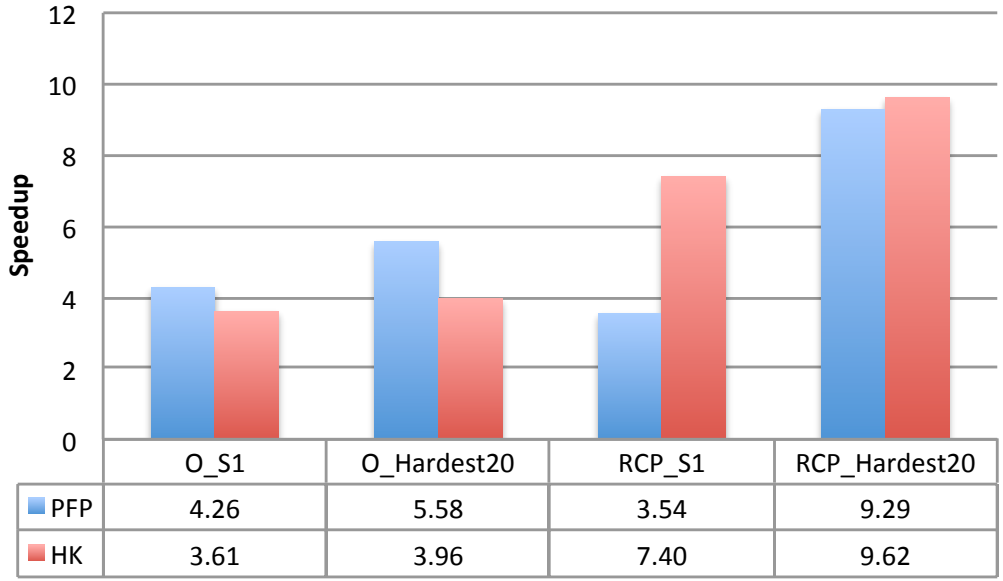
(c) Original graphs



(d) Permuted graphs

- (x, y): with y probability, the corresponding algorithm obtains a performance that is at most x times worse than the best runtime.
- The plots clearly mark the GPU algorithm as the fastest in most cases,
- The GPU algorithm obtains the best performance in 61% of the original graphs and in 74% of the permuted ones.

Overall speedup



	O_S1	O_Hardest20	RCP_S1	RCP_Hardest20
PFP	4.26	5.58	3.54	9.29
HK	3.61	3.96	7.40	9.62

Figure: GPU algorithm w.r.t. PFP (left bars) and HK (right bars) algorithms.

- Average speedup: 3.61 and 3.54 on original and permuted graphs.
- Hardest instances: 3.96 and 9.29 on original and permuted graphs.
- Robust running time (execution times for different repetitions)
  - For O_S1, the ratios of the standard deviations to the average time are less than 10%, 18%, and 47% for 20, 5, and 3 graphs.

Concluding remarks

- GPU implementation of a BFS-based maximum cardinality matching algorithm for bipartite graphs.
- The experiments showed that the GPU implementation is faster than the existing multicore implementations.
- The speedups achieved with respect to well-known sequential implementations varied from 0.03 to 629.19, averaging 9.29 w.r.t. the fastest sequential algorithm on a set of 20 hardest problems.
- Everything was on the GPU; a limited memory device. Thinking about what can be done for graphs that does not fit into a GPU.

Further information

Thank you for your attention.

`http://perso.ens-lyon.fr/bora.ucar`

Actual running time

	Original graphs				Permuted graphs			
Matrix name	GPU	P-DBFS	PFP	HK	GPU	P-DBFS	PFP	HK
roadNet-CA	**0.34**	0.53	0.95	2.48	**0.39**	1.88	3.05	4.89
delaunay_n23	**0.96**	1.26	2.68	1.11	**0.90**	5.56	3.27	14.34
coPapersDBLP	**0.42**	6.27	3.11	1.62	0.38	1.25	**0.29**	1.26
kron_g500-logn21	**0.99**	1.50	5.37	4.73	**3.71**	4.01	64.29	16.08
amazon-2008	**0.11**	0.18	6.11	1.85	**0.41**	1.37	61.32	4.69
delaunay_n24	**1.98**	2.41	6.43	2.22	**1.86**	12.84	6.92	35.24
as-Skitter	**0.49**	1.89	7.79	3.56	**3.27**	5.74	472.63	29.63
amazon0505	**0.18**	22.70	9.05	1.87	**0.24**	15.23	17.59	2.23
wikipedia-20070206	**1.09**	5.24	11.98	6.52	**1.05**	5.99	9.74	5.73
Hamrle3	1.36	2.70	**0.04**	12.61	**3.85**	7.39	37.71	57.00
hugetrace-00020	**7.90**	393.13	15.95	15.02	**1.52**	9.97	8.68	38.27
hugebubbles-00000	13.16	**3.55**	19.81	5.56	**1.80**	10.91	10.03	38.97
wb-edu	33.82	8.61	**3.38**	20.35	17.43	20.10	**9.49**	51.14
rgg_n_2_24_s0	3.68	2.25	25.40	**0.12**	**2.20**	12.50	5.72	31.78
patents	0.88	**0.84**	92.03	16.18	**0.91**	0.97	101.76	18.30
italy_osm	5.86	1.20	**1.02**	122.00	**0.70**	3.97	6.24	18.34
soc-LiveJournal1	**3.32**	14.35	243.91	21.16	**3.73**	7.14	343.94	20.71
ljournal-2008	**2.37**	10.30	360.31	17.66	**6.90**	7.58	176.69	23.45
europe_osm	57.53	**11.21**	14.15	1911.56	**7.21**	37.93	68.18	197.03
com-livejournal	**4.58**	22.46	2879.36	34.28	**5.88**	17.19	165.32	29.40

Except six among the original graphs and another two among the permuted graphs, the GPU algorithm is faster than the best sequential algorithm. It is also faster than the multicore ones in all, except five original graphs.

References

Deveci, M., Kaya, K., Çatalyürek, Ü. V., and Uçar, B.:

A push-relabel-based maximum cardinality matching algorithm on GPUs,
ICPP2013.

Azad, A., Halappanavar, M., Rajamanickam, S., Boman, E.G., Khan, A., Pothen, A.:

Multithreaded algorithms for maximum matching in bipartite graphs. In:
26th IPDPS. pp. 860–872. IEEE (2012)

Duff, I.S., Kaya, K., Uçar, B.:

Design, implementation, and analysis of maximum transversal algorithms.
ACM TOMS 38(2), 13 (2011)

Fagginger Auer, B., Bisseling, R.:

A GPU algorithm for greedy graph matching.
Facing the Multicore-Challenge II pp. 108–119 (2012)

Halappanavar, M., Feo, J., Villa, O., Tumeo, A., Pothen, A.:

Approximate weighted matching on emerging manycore and multithreaded architectures.
Int. J. High Perform. C. 26(4), 413–430 (2012)

Kaya, K., Langguth, J., Manne, F., Uçar, B.:

Push-relabel based algorithms for the maximum transversal problem.
Comput. Oper. Res. 40(5), 1266–1275 (2012)

Vasconcelos, C., Rosenhahn, B.:

Bipartite graph matching computation on GPU.
In: Energy Minimization Methods in Computer Vision and Pattern Recognition. pp. 42–55. (2009)