TR-2014-12

A Hybrid GPU-CPU Parallel CM Reordering Algorithm for Bandwidth Reduction of Large Sparse Matrices

Ang Li, Radu Serban, Dan Negrut

May 24, 2015

Abstract

We present a hybrid GPU-CPU implementation of a reordering strategy for permuting elements to make the half-bandwidth of a sparse matrix lower. This algorithm, based on work by Cuthill and McKee [4], is a critical component of the SPIKE-based preconditioner provided by the SaP::GPU library [3]. We discuss the three stages of the unordered implementation of the Cuthill-McKee (`CM`) algorithm and compare our implementation against both a self-implemented CPU `CM` algorithm and the `MC60` algorithm provided by the HSL library [1], which also targets reducing the half-bandwidth. Numerical experiments, performed on around 130 matrices arising in various engineering and scientific applications, indicate that our implementation almost obtains similar or narrower reordered matrix, and for large matrices with dimension over 10^6 and number of nonzero entries over 10^7, our implementation is more efficient than both other implementations.

Contents

1 Introduction

Symmetric reordering operations that reduce half-bandwidth[1] of a sparse matrix are useful in a variety of applications. In particular, such reorderings are a crucial element for a linear solver whose performance is dependent on both the dimension and the bandwidth of a matrix, for instance Math Kernel Library's banded matrix solver. Given a sparse matrix, the problem of finding a reordering that results in the lowest half-bandwidth is NP-hard. Practically speaking, however, it is not necessary to aim at targeting the lowest half-bandwidth. Cuthill-McKee (`CM`), which is a variant of the standard Breadth-First Search (`BFS`) algorithm, gives a simple and efficient heuristic in reducing the half-bandwidth. As `CM` reordering is symmetric, it can follow reorderings aiming at increasing the absolute values of diagonal entries, e.g., the algorithms proposed by Duff and Koster [6], without disturbing the results of the previous reordering. These permutations together lead to banded matrices with low bandwidth and high degree of diagonal dominance[2].

This report documents details of our implementation of the unordered variant of the algorithm proposed by Karantasis et al. [2]. We compare this implementation against our own CPU implementation and the Harwell Standard Library (HSL) [1] algorithm `MC60`, in terms of both efficiency and quality (i.e. the resulting half-bandwidth). Numerical experiments, performed on around 130 matrices arising in various engineering and scientific applications, indicate that our implementation almost obtains similar or narrower reordered matrix, and for large matrices with dimension over 10^6 and number of nonzero entries over 10^7, our implementation is more efficient than both other implementations.

The remainder of this report is organized as follows. In §2, we provide a brief overview of the `CM` algorithm [4]. The three stages of the implementation are described in detail in §3. Numerical experiments and results are presented in §4. In §5 we discuss potential further improvements and various possibilities of parallelizing our implementation. Finally, §6 summarizes our findings. For convenience, in the next sections, our hybrid and sequential CPU implementations are denoted as `HYB` and `SEQ`, respectively. Also, we use `MC60` for HSL's `MC60`, and denote half-bandwidth as K and execution time as T in all tables.

2 Reordering algorithm description

As stated in §1, the problem of finding the potential lowest half-bandwidth by symmetrically reordering a sparse matrix is NP-hard. `CM`, on the other hand, gives a simple and efficient heuristic. In this section, the matrix $\mathbf{A}$ we are talking about is assumed to be symmetric and connected. Given a non-symmetric matrix $\mathbf{A}$, $\frac{\mathbf{A}+\mathbf{A}^T}{2}$ is the matrix to work on. If $\mathbf{A}$ contains two or more connected components, those components should be separated and we work on each of them.

[1] The half-bandwidth of a matrix $\{a_{ij}\}$ is defined as $\max_{i,j,a_{ij}\neq 0} |i-j|$.

[2] The degree of diagonal dominance of a matrix $\{a_{ij}\}$ is defined as $d = \min_j \sup d_j$ where $|a_{ii}| \geq d_j \cdot \sum_{i\neq j} |a_{ij}|$.

Given a symmetric and connected $n \times n$ matrix $\mathbf{A}$ with m non-zero entries, we visualize it as the adjacency matrix of a graph. `CM` first picks a random node and adds the node to the worklist. Then the algorithm repeats the following until all nodes have been added once to the worklist:

- Pick the first node n_0 in the worklist and remove it;
- Find all neighboring nodes of n_0 which have not been added to the worklist yet, and these nodes form a list L;
- Sort L with ascending vertex degree.

In other words, `CM` is essentially a `BFS` where neighboring vertices are visited in order from lowest to highest vertex degree. Note that from the assumption of matrix $\mathbf{A}$, this graph contains only one connected component. This implies that before the algorithm completes (i.e. all nodes have been added to the worklist once), the first step of the loop above is always valid.

With different selections of the starting nodes, the height of the `BFS` tree will also differ, which affects the quality of `CM` (i.e. the half-bandwidth of resulting matrix). Heuristically speaking, the taller the `BFS` tree is, the fewer nodes two adjacent levels contain, and the lower the half-bandwidth will be (note that nodes at two levels which are not adjacent are not connected, and thus do not influence the half-bandwidth). Thus the selection of the starting node is crucial. We improve the selection of the starting node in two ways: starting from a node with lowest vertex degree, and working on several starting nodes. We will discuss this topic in detail in §3.2.

3 Implementation details

Our implementation of the algorithm is called unordered `CM` algorithm, which is described in [2]. This implementation is separated into three stages:

Stage 1 – Pre-process to get the matrix to work on,

Stage 2 – Pick the starting node and do a pass of standard `BFS` to get levels for all nodes in the `BFS` tree,

Stage 3 – Reorder the nodes to obtain final permutation.

In these three stages, Stage Two requires trials on several nodes for a high-quality `CM` execution (i.e. smaller half-bandwidth), which means several iterations of `BFS` are expected. In the remainder of the report, we call each iteration in Stage Two a "`CM` iteration". In our implementation, Unified Memory, a new feature of CUDA 6.0, is leveraged so that whenever the workspace is switched between CPU and GPU, no explicit data transfer is required, which results in simpler memory management and a smaller code length.

3.1 Stage 1: Pre-processing

In the first stage, we would like to achieve the following two objectives:

- Generate the symmetric matrix we are working on. As the input matrix $\mathbf{A}$ is not guaranteed to be symmetric, all row and column indices of non-zero entries in matrix $\frac{\mathbf{A}+\mathbf{A}^T}{2}$ are generated, so that in the next two stages we are working on a symmetric matrix.

- Pre-sorting all non-zero entries. In `CM` algorithm, it is expensive to repetitively sort the neighbors of the current node as stated in §2. Pre-sorting the nodes with the same row indices by ascending vertex degree of column indices removes this necessity with the extra cost of a single sort.

As both generating the non-zero entries and pre-sorting are highly parallizable, this stage is implemented on GPU.

3.2 Stage 2: Picking the starting node and standard `BFS`

As described in §2, there are two methodologies which are helpful in picking the start node: picking the node with the smallest vertex degree, and starting from several nodes. By starting from the node with smallest vertex degree, at the second level of the `BFS` tree there will be fewer nodes. Eventually the resulting `BFS` tree is more inclined to be "tall and thin". By starting from several nodes and completing Stage Two and Stage Three for each of them, the possibility of starting from a "bad" node decreases.

After selecting the starting node, we do a pass of standard `BFS` to obtain the levels of all nodes in the `BFS` tree. Since the order of nodes at the same level is not critical in this stage, we see the potential of parallelism in concurrently visiting the neighbors of all nodes at the previous level. We use an outer loop to iterate over the levels, and in each iteration, depending on the number of nodes added in the previous iteration (denoted as n_p), we decide whether this iteration is executed on GPU or CPU. Heuristically speaking, the larger n_p is, the larger the number of their neighbors is, the more performance, instead of overheads, a GPU kernel call can bring us. The threshold we set for using CPU or GPU is 10: if $n_p \geq 10$, a kernel is called; otherwise we stay on CPU.

There are two issues in the implementation details of Stage Two:

- By starting from several nodes, we are not running the algorithms concurrently on several randomly selected nodes. Instead, `CM` iterations get executed sequentially. After each iteration, the node at the last level with the lowest vertex degree which has not yet been attempted as starting node is selected. If no such nodes exist (i.e. all nodes at the last level have been attempted), a random node which has not been attempted is selected.

- `CM` iterations terminate when the height of the `BFS` tree does not increase OR the maximum number of nodes over all levels does not decrease compared with the optimal

solution we have already found. This idea comes from the Ph.D. thesis [7] with the difference that we only bother looking at the leaf with the minimum degree. From §4, we observe that with this stopping criterion, for most matrices, the algorithm terminates within three `CM` iterations while it can obtain half-bandwidth no larger than HSL's `MC60`.

3.3 Stage 3: Reordering the nodes

After Stage Two, the algorithm has found the level for each node. Let us denote the maximum level as ml. Note that coarsely speaking, nodes are expected to be ordered with ascending level from level 0 up to level ml, and a fixed amount of space can be pre-allocated for nodes at each level (and this fact sheds light upon when all nodes at a certain level are processed). On a different observation, since the order of nodes at level l only directly depends on the order of nodes at level $l-1$, we can use this level-grained parallelism in the following way: a pair of read/write pointers is set for each level, and except for level 0, the read/write pointers of each level will point to the starting position of the level's pre-allocated space. We say a thread "works on" level l if it reads nodes at level l and writes their neighbors that are at level $l+1$. Thus the thread working on level l will read and modify the read pointer of level l and the write poiner of level $l+1$, and it will only read the write pointer of level l. Once the thread finishes reading all nodes at level l, it moves on to another level; otherwise it repeats checking whether the thread working on level $l-1$ has written nodes which it has not processed or not. If yes, the thread working on level l processes these nodes (i.e. write their neighbors with level $l+1$), and goes back to check again whether it has finished processing or not; otherwise, it spins and waits for the thread working on the previous level.

The parallelism of this algorithm stated in this subsection is rather coarse-grained and is SIMT-typed, which will perform badly on GPU, which is better at fine-grained and SIMD-typed parallelism.

4 Evaluation

Performance of the proposed implementation was evaluated on a set of 130 sparse matrices from various applications. Most of these matrices were selected from the Florida Sparse Matrix Collection [5]. In addition, matrices ANCF31770, ANCF88950, ancfBigDan and NetANCF_40by40 arise in the implicit integration of large-scale flexible body dynamics [8]. Tables 1, 2 and 3 show the dimensions and numbers of nonzero entries.

The comparisons were based on the following metrics:

- Obtained half-bandwidth;
- Running time required to reorder, with separate times reported for each of the three individual stages;

Note that the dimensions of these matrices vary. For all results, we partition them into 13 larger matrices and 117 smaller matrices. Moreover, three "smaller" matrices (boy1d, ASIC_100k and HTC_336_4438)are not displayed in these figures as the speedup we obtained over HSL's `MC60` is significantly more than other matrices.

The simulation results are listed in Tables 4, 5 and 6 with all timings in milliseconds.

Half-bandwidth. Fig. 1 and Fig. 2 display the comparison of half-bandwidth obtained by `MC60` and `HYB` for each matrix. Fig. 3 and Fig. 4 display the comparison of half-bandwidth obtained by `SEQ` and `HYB` for each matrix. The value is obtained by calculating the half-bandwidth obtained by `MC60` or `SEQ` over the half-bandwidth obtained by `HYB`. This implies the larger this value is, the higher quality our implementation can achieve. From these four figures, we can see for almost all matrices `HYB` can obtain similar or lower half-bandwidths. For around 20 matrices, `HYB` is able to obtain significantly lower half-bandwidths, compared with both other implementations.

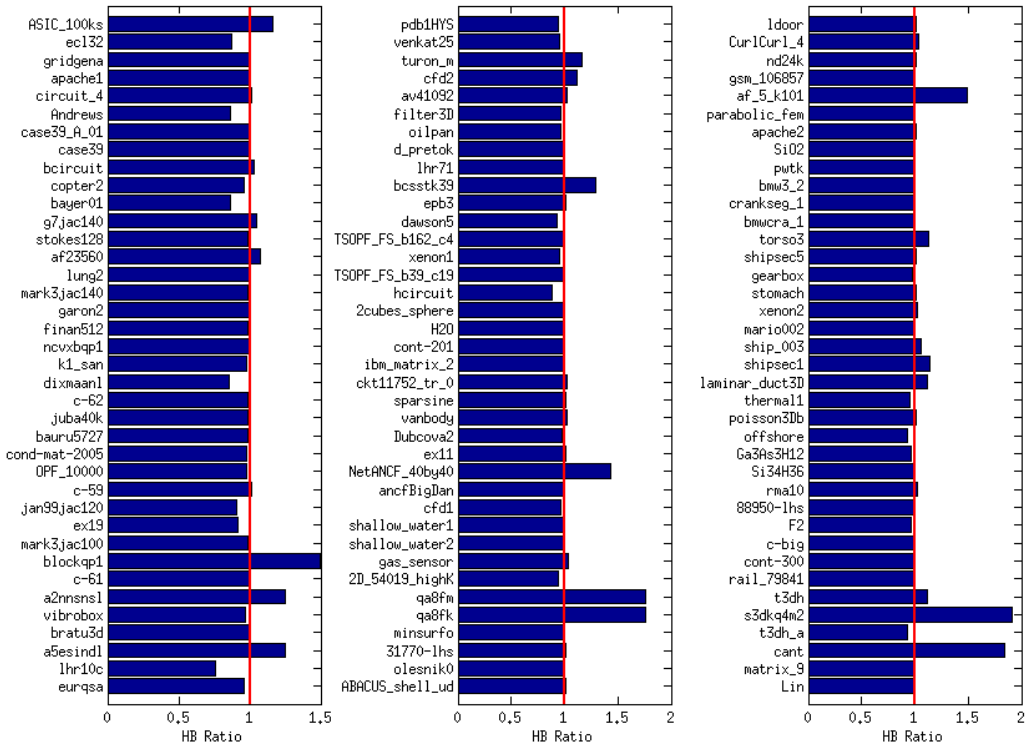

Figure 1: Half bandwidth comparison between `HYB` and `MC60` for smaller matrices. Note that the value is defined as $\frac{K_{MC60}}{K_{HYB}}$, thus the larger the better.

Performance. Fig. 5 and Fig. 6 display the speedup of `HYB` over `MC60`. Fig. 7 and Fig. 8 display the speedup of `HYB` over `SEQ`. The results are twofold: for matrices with larger

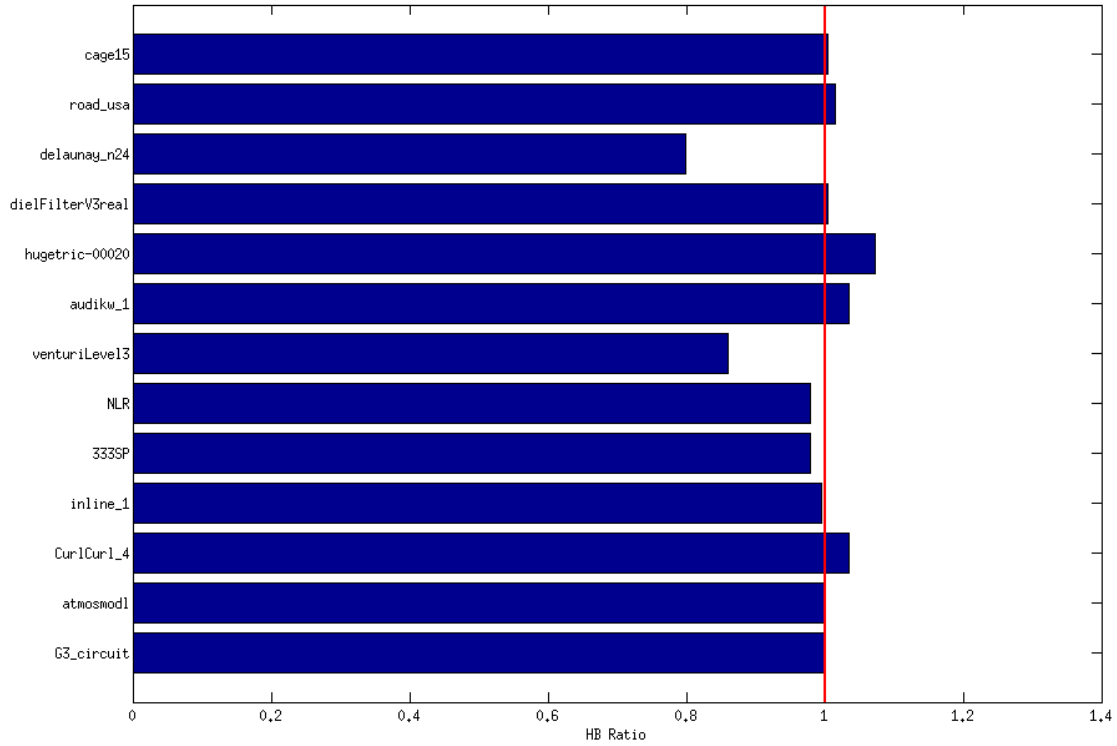

Figure 2: Half bandwidth comparison between `HYB` and `MC60` for larger matrices, the larger the better.

dimensions and a larger number of nonzeros, in most cases `HYB` will outperform both other implementations (which can be observed from Fig. 6, Fig. 8 and the third column of Fig. 5 and Fig. 7, which show matrices whose execution time is rather long); for smaller matrices, however, `MC60` and `SEQ` in most cases run faster than `HYB`. This matches our expectation and can be explained as follows: when the dimension and number of nonzeros are small, the potential degree of parallelism is rather small so that the parallel sections (either on GPU or the CPU OpenMP part) cannot pay back all overheads, including the overheads of data transfer between CPU and GPU, the overheads of GPU kernel synchronization and OpenMP thread synchronization.

Fig. 9 and Fig. 10 show the breakdown of execution time over the three algorithmic stages of `HYB` for all matrices. The conclusions are : for almost all matrices, either Stage One (i.e. Pre-processing) or Stage Two (i.e. standard BFS) will consume most execution time. To some extent, this meets our expectation. The time complexity of pre-sorting in Stage One is in the order of $O(NNZ \cdot \log(NNZ))$ (where NNZ is the number of non-zero entries in this matrix). For Stage Two, to select the starting point, several iterations of standard BFS has to be executed. Furthermore, as for each level, a synchronization is required. This fact leads

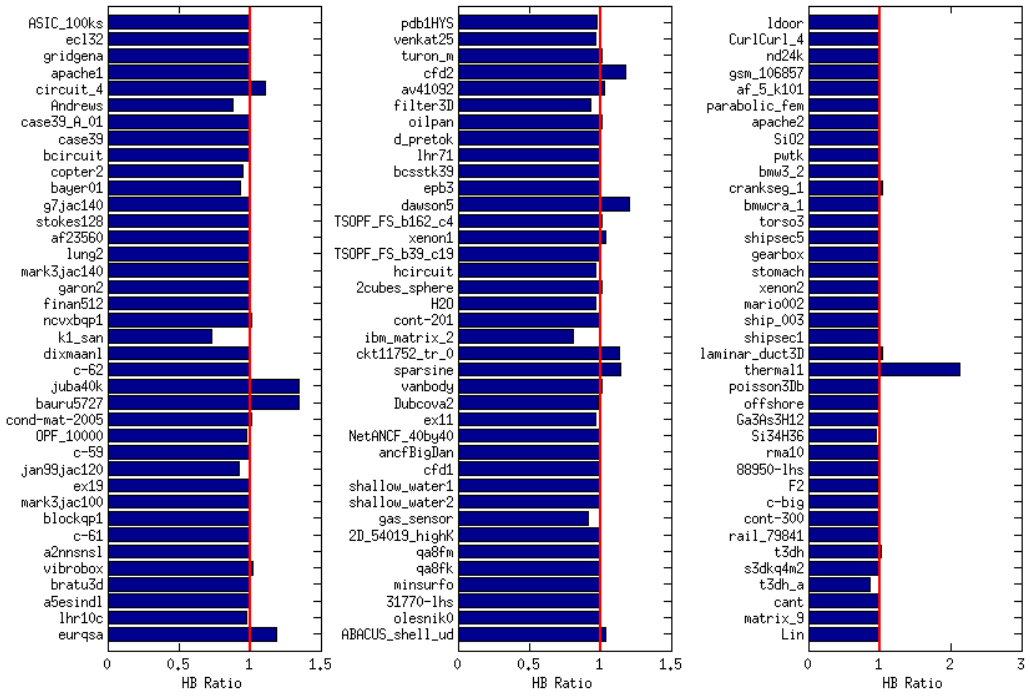

Figure 3: Half bandwidth comparison between `HYB` and `SEQ` for smaller matrices. Note that the value is defined as $\frac{K_{SEQ}}{K_{HYB}}$, thus the larger the better.

to the limitation of the speedup of the whole algorithm. For Stage Three, as long as the starting point is selected, we do only one pass of node placement, which can be parallelized among levels.

5 Future Work

Since `HYB` does not always perform well, we plan to keep both `SEQ` and `HYB`. It is worthwhile to investigate a simple heuristic to figure out in which cases `HYB` should be used instead of `SEQ`.

We also see the potential to improve performance if we move the sorting from Stage One to Stage Three (and this means we will do the sorting on CPU). The global sort can thus break into sorts on small segments, and therefore less work needs to be done on sorting.

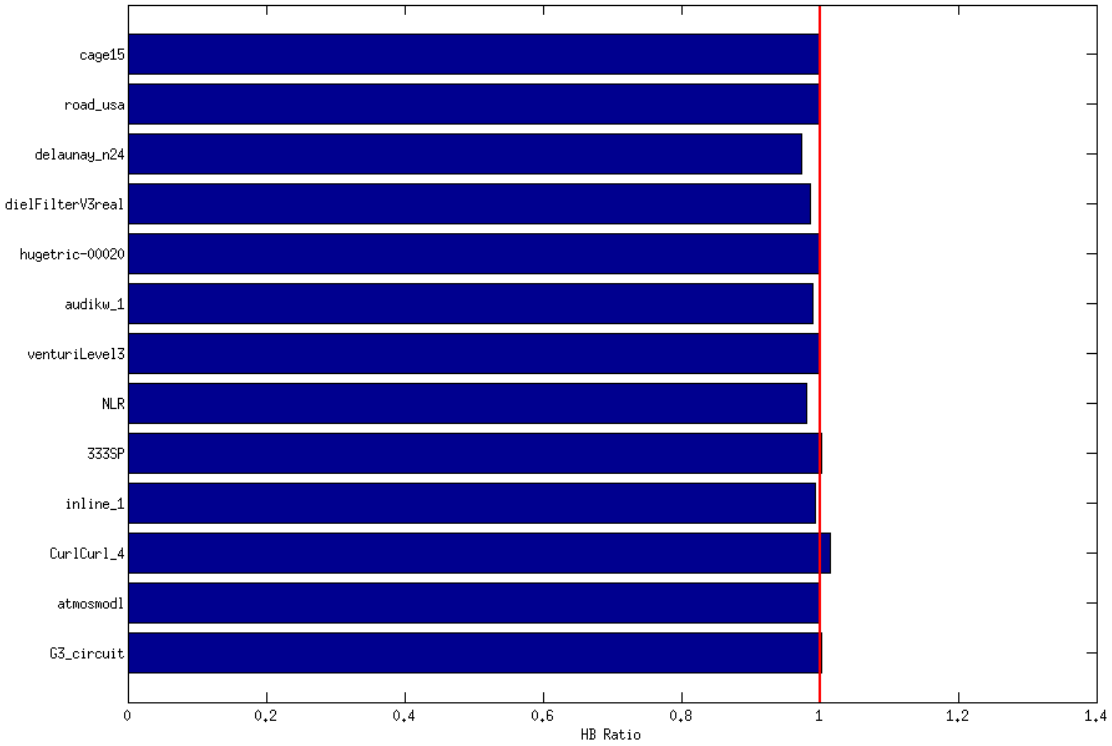

Figure 4: Half bandwidth comparison between `HYB` and `SEQ` for larger matrices, the larger the better.

6 Conclusions

This report discusses an implementation of an algorithm that seeks to reduce the half-bandwidth of a sparse matrix. The software implementation of the algorithm is compared in terms of speed and quality with the HSL `MC60` and a self-implemented CPU `CM` algorithm. Our findings are as follows:

- Compared with both `MC60` and the sequential implementation, the hybrid implementation is able to obtain high quality of ordering in terms of the resulting half-bandwidth for almost all application matrices.
- Compared with both `MC60` and the sequential implementation, whose parallelism (if any) is among levels and highly constrained, the hybrid implementation can obtain good performance when dealing with a large matrix which is less sparse.
- The hybrid implementation, though it is efficient in reducing half-bandwidth of a sparse matrix, is not always a silver bullet in terms of seeking for good performance when the matrix size is small.

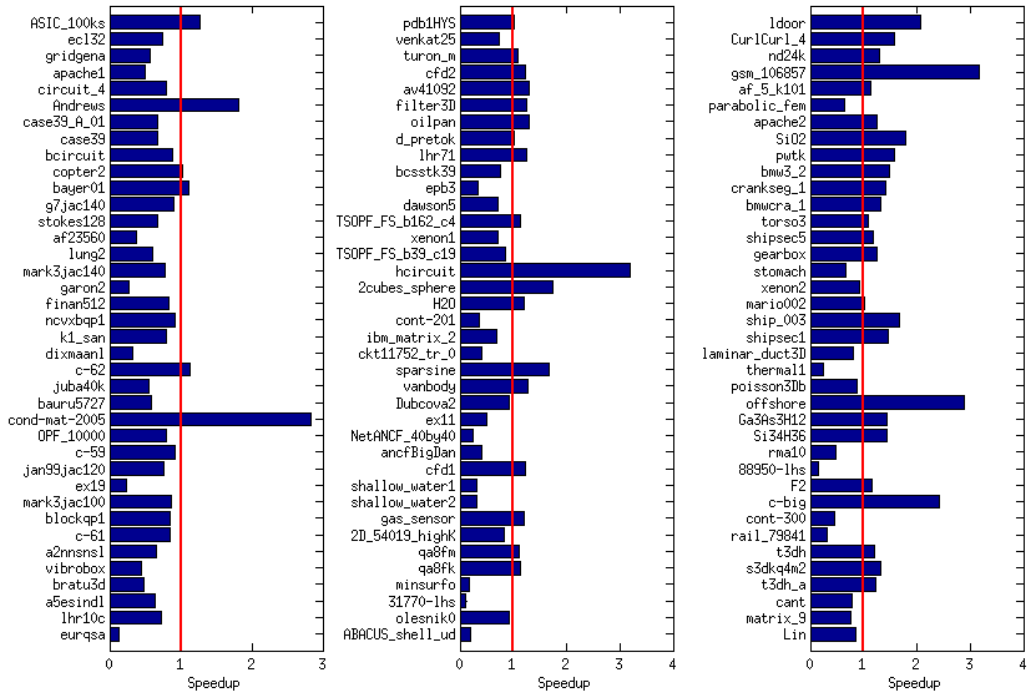

Figure 5: Speedup of `HYB` over `MC60` for smaller matrices, the larger the better.

- An attempt to move the entire solution onto the GPU slowed the overall algorithm implementation tremendously. Specifically, owing to the fact that the level of parallelism in task Stage Three is rather coarse, we are always worse off keeping this stage on GPU.
- We leveraged Unified Memory, a new feature of CUDA 6.0, in our hybrid implementation. This helps us achieve a simpler memory management and a smaller code length.

References

[1] HSL: A collection of Fortran codes for large-scale scientific computation. http://www.cse.clrc.ac.uk/nag/hsl, 2011.

[2] Parallel Reordering: Graph processing on the dark side of BFS. http://i2pc.cs.illinois.edu/public_archive/karantasis-i2pc.pdf, 2013.

[3] SPIKE GPU: An implementation of a recursive divide-and-conquer parallel strategy for solving large systems of linear equations. http://spikegpu.sbel.org, 2013.

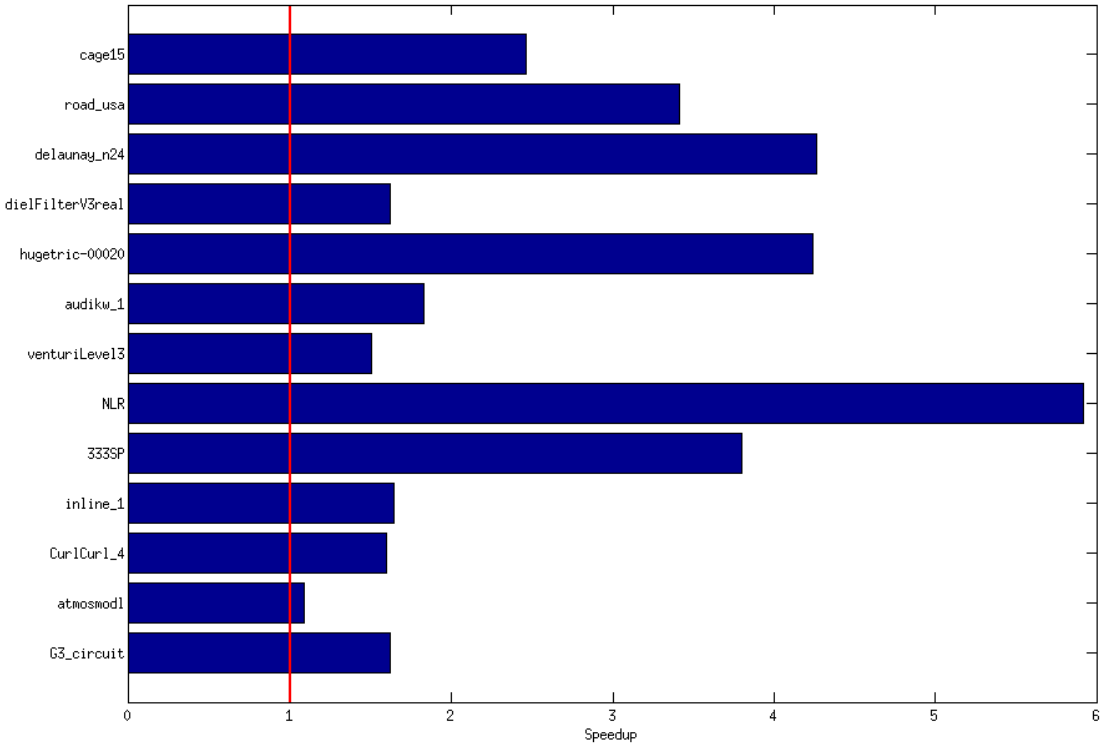

Figure 6: Speedup of `HYB` over `MC60` for larger matrices, the larger the better.

[4] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

[5] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[6] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[7] D. A. Padua Haiek. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Champaign, IL, USA, 1980. AAI8018194.

[8] R. Serban, D. Melanz, A. Li, I. Stanciulescu, P. Jayakumar, and D. Negrut. A GPU-based preconditioned Newton-Krylov solver for flexible multibody dynamics. *Int. J. Num. Meth. Eng.*, 2014. submitted.

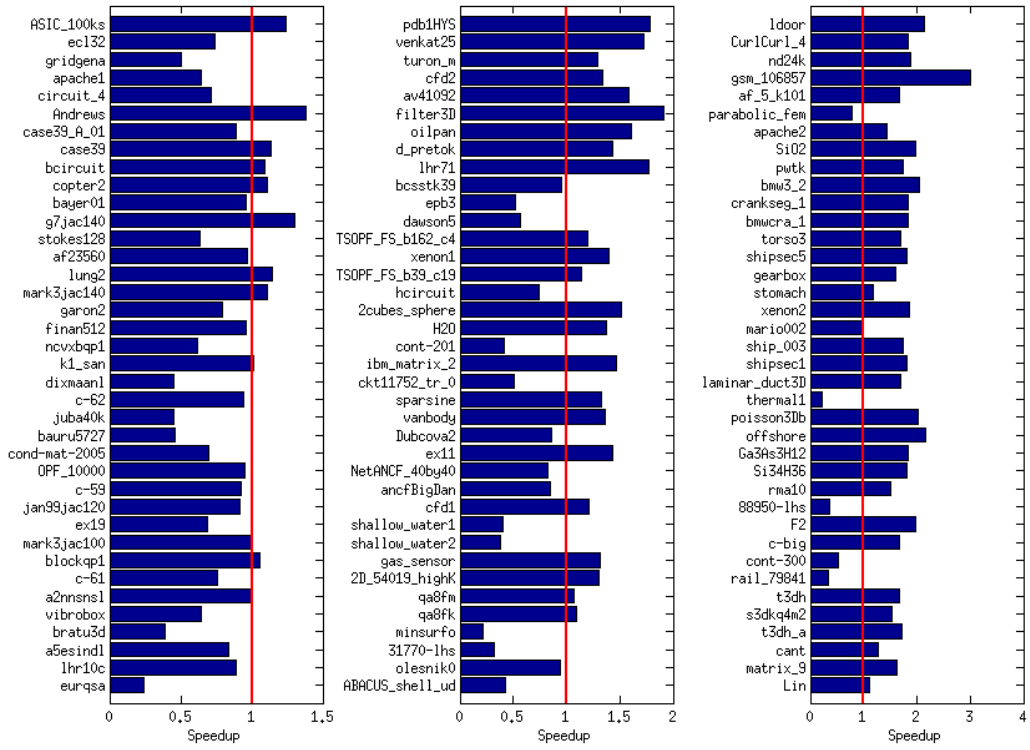

Figure 7: Speedup of `HYB` over `SEQ` for smaller matrices, the larger the better.

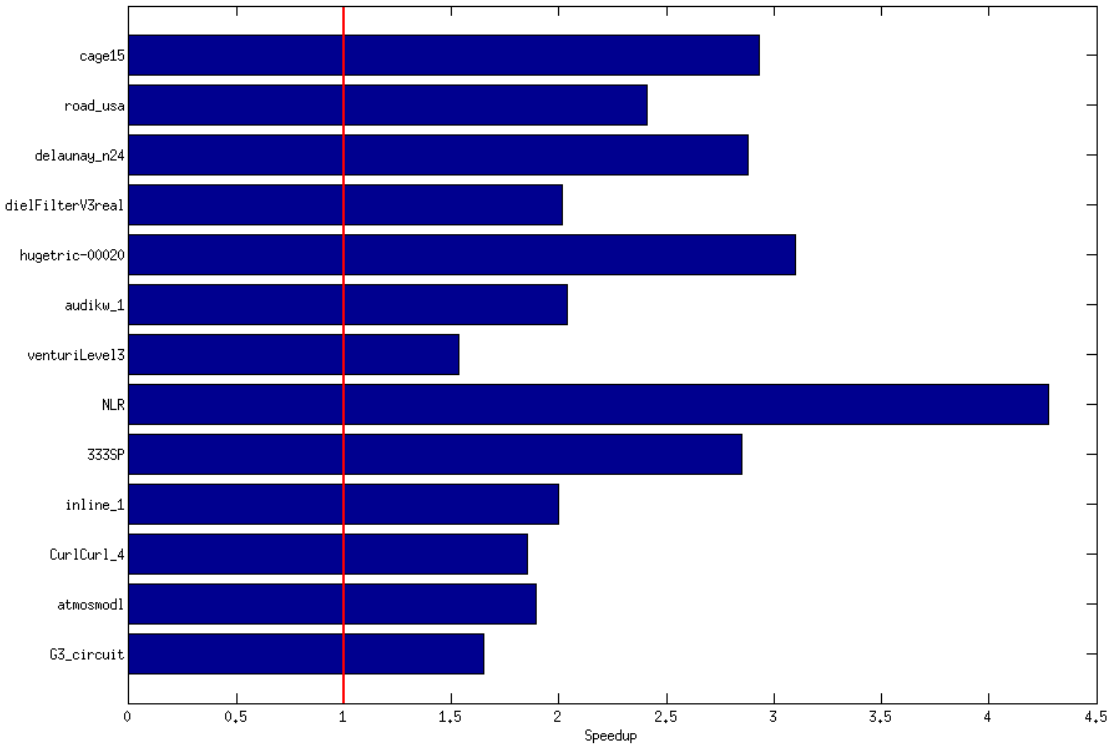

Figure 8: Speedup of `HYB` over `SEQ` for larger matrices, the larger the better.

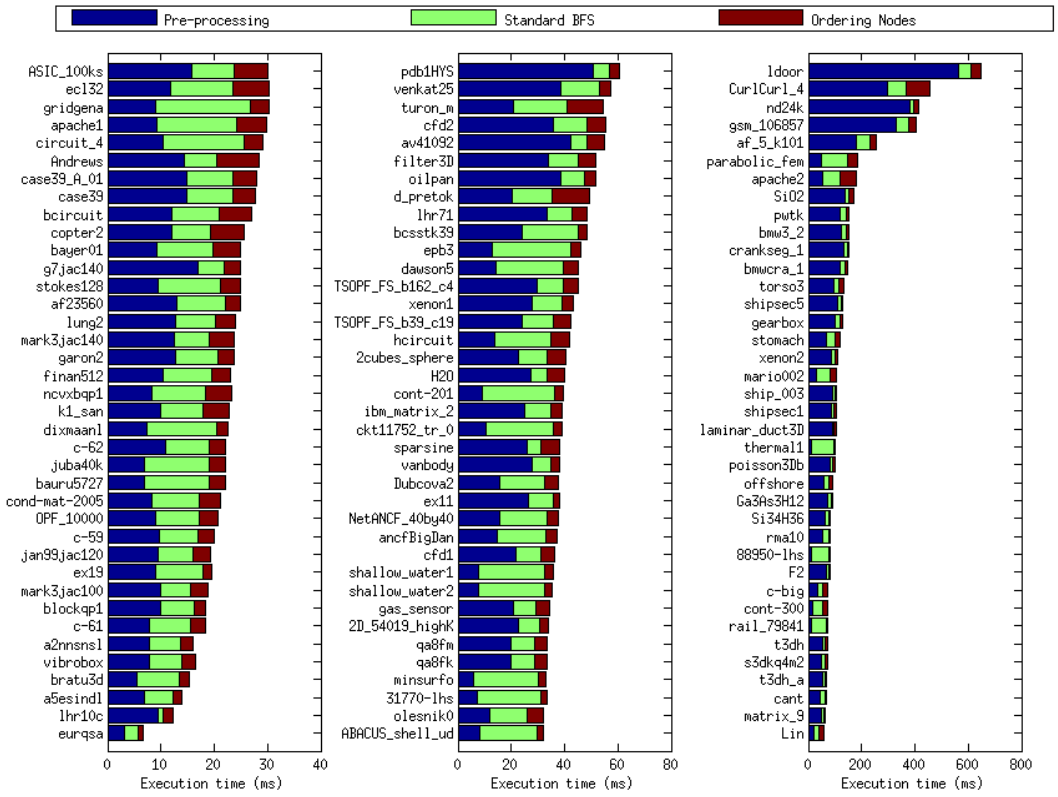

Figure 9: Breakdown of execution time over the three algorithmic stages of `HYB` for smaller matrices.

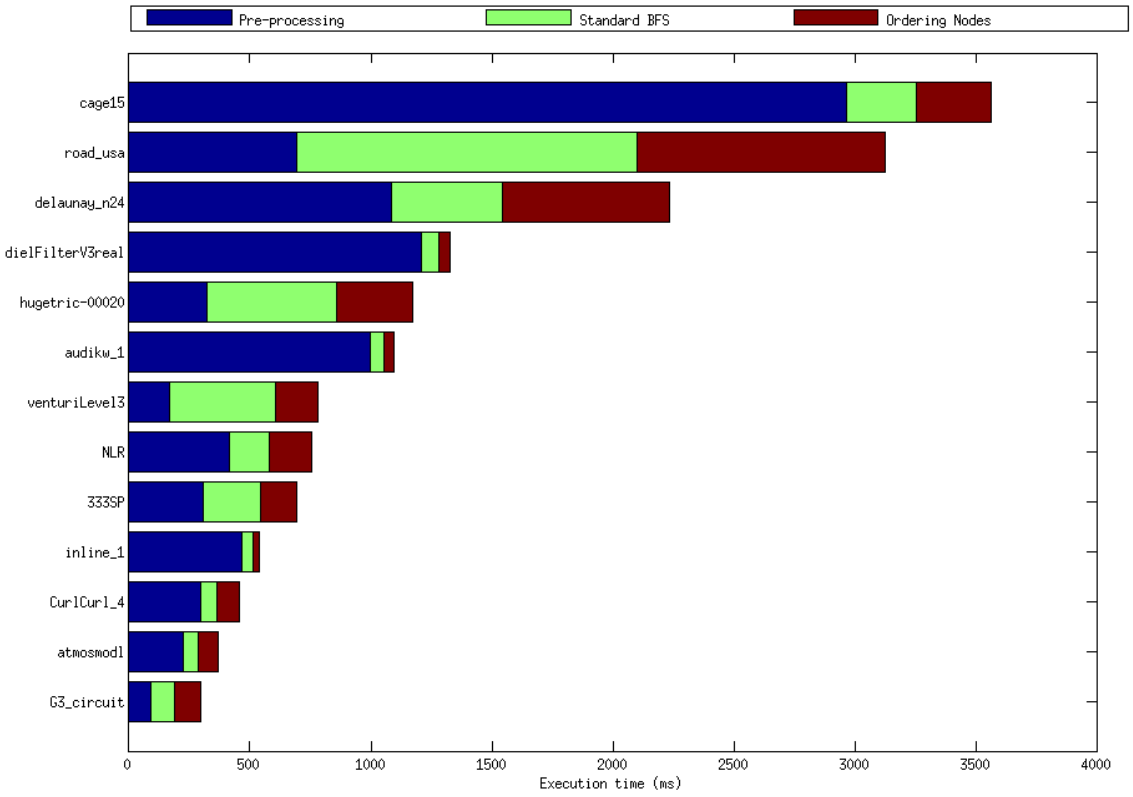

Figure 10: Breakdown of execution time over the three algorithmic stages of `HYB` for larger matrices.

Table 1: Information of test sparse matrices (1/3).

Mat. No	Mat. Name	Dimension	No. of Non-Zeros	Original K
1	2cubes_sphere	101,492	874,378	100,407
2	2D_54019_highK	54,019	996,414	53,964
3	31770-lhs	31,770	183,540	26,544
4	333SP	3,712,815	222,217,266	3,681,176
5	88950-lhs	88,950	513,900	74,244
6	a2nnsnsl	80,016	196,115	75,014
7	a5esindl	60,008	145,004	60,006
8	ABACUS_shell_ud	23,412	218,484	379
9	af_5_k101	503,625	9,027,150	859
10	af23560	23,560	484,256	304
11	ancfBigDan	63,603	569,262	62,889
12	Andrews	60,000	410,077	59,925
13	apache1	80,800	311,492	808
14	apache2	715,176	2,766,523	65,837
15	ASIC_100k	99,340	954,163	99,106
16	ASIC_100ks	99,190	578,890	98,439
17	atmosmodl	1,489,752	10,319,760	39,204
18	audikw_1	943,695	39,297,771	925,946
19	av41092	41,092	1,683,902	31,055
20	bauru5727	40,366	145,019	18,275
21	bayer01	57,735	277,774	57,733
22	bcircuit	68,902	375,558	67,086
23	bcsstk39	46,772	1,068,033	555
24	blockqp1	60,012	340,022	40,012
25	bmw3_2	227,362	5,757,996	219,110
26	bmwcra_1	148,770	5,396,386	141,050
27	boyd1	93,279	652,246	93,269
28	bratu3d	27,792	88,627	26,687
29	c-59	41,282	260,909	41,281
30	c-61	43,618	176,817	43,617
31	c-62	41,731	300,537	41,690
32	c-big	345,241	1,343,126	338,366
33	cage15	5,154,859	99,199,551	2,321,683
34	cant	62,451	2,034,917	275
35	case39	40,216	526,139	40,206
36	case39_A_01	40,216	526,139	40,206
37	cfd1	70,656	949,510	6,229
38	cfd2	123,440	1,605,669	4,501
39	circuit_4	80,209	307,604	80,193
40	ckt11752_tr_0	49,702	333,029	49,452
41	cond-mat-2005	40,421	175,693	40,349
42	cont-201	80,595	239,596	69,996
43	cont-300	180,895	539,396	157,496
44	copter2	55,476	407,714	55,279

Table 2: Information of test sparse matrices (2/3).

Mat. No	Mat. Name	Dimension	No. of Non-Zeros	Original K
45	crankseg_1	52,804	5,333,507	50,388
46	CurlCurl_4	2,380,515	26,515,867	43,681
47	d_pretok	182,730	885,416	129,917
48	dawson5	51,537	531,157	19,383
49	delaunay_n24	16,777,216	50,331,601	16,769,102
50	dielFilterV3real	1,102,824	45,204,422	1,036,475
51	dixmaanl	60,000	179,999	40,000
52	Dubcova2	65,025	547,625	64,820
53	ecl32	51,993	380,415	51,840
54	epb3	84,617	463,625	284
55	eurqsa	7,245	25,633	5,964
56	ex11	16,614	1,096,948	4,839
57	ex19	12,005	259,879	169
58	F2	71,505	2,682,895	71,309
59	filter3D	106,437	1,406,808	8,276
60	finan512	74,752	335,872	74,724
61	G3_circuit	1,585,478	4,623,152	947,128
62	g7jac140	41,490	565,956	15,047
63	Ga3As3H12	61,349	3,016,148	20,893
64	GaAsH6	61,349	1,721,579	20,005
65	garon2	13,535	390,607	13,531
66	gas_sensor	66,917	885,141	2,901
67	gearbox	153,746	4,617,075	137,410
68	gridgena	48,962	280,523	405
69	gsm_106857	589,446	11,174,185	588,744
70	H2O	67,024	1,141,880	14,976
71	hcircuit	105,676	513,072	105,663
72	HTC_336_4438	226,340	565,431	94,690
73	hugetric-00020	7,122,792	10,680,777	7,122,285
74	ibm_matrix_2	51,448	1,056,610	51,445
75	inline_1	503,712	18,660,027	502,403
76	jan99jac120	41,374	260,202	13,495
77	juba40k	40,337	144,945	18,246
78	k1_san	67,759	303,364	48,251
79	laminar_duct3D	67,173	3,833,077	52,173
80	ldoor	952,203	23,737,339	686,979
81	lhr10c	10,672	232,633	8,419
82	lhr71	70,304	1,528,092	19,503
83	Lin	256,000	1,011,200	6,400
84	lung2	109,460	492,564	107,141
85	mario002	389,874	1,167,685	387,647
86	mark3jac100	45,769	285,215	4,075
87	mark3jac140	64,089	399,735	4,075
88	matrix_9	103,430	2,121,550	103,429

Table 3: Information of test sparse matrices (3/3).

Mat. No	Mat. Name	Dimension	No. of Non-Zeros	Original K
89	minsurfo	40,806	122,214	202
90	ncvxbqp1	50,000	199,984	33,333
91	nd24k	72,000	14,393,817	68,114
92	NetANCF_40by40	63,603	569,262	62,889
93	NLR	4,163,763	24,975,952	4,163,523
94	offshore	259,789	2,251,231	237,738
95	oilpan	73,752	1,835,470	3,877
96	olesnik0	88,263	402,623	64,453
97	OPF_10000	43,887	255,799	42,825
98	parabolic_fem	525,825	2,100,225	525,820
99	pdb1HYS	36,417	2,190,591	34,064
100	poisson3Db	85,623	2,374,949	85,563
101	pwtk	217,918	5,926,171	189,331
102	qa8fk	66,127	863,353	1,048
103	qa8fm	66,127	863,353	1,048
104	rail_79841	79,841	316,881	79,811
105	rajat26	51,032	249,302	50,960
106	rma10	46,835	2,374,001	25,141
107	road_usa	23,947,347	28,854,312	22,895,468
108	s3dkq4m2	90,449	2,455,670	614
109	shallow_water1	81,920	204,800	40,959
110	shallow_water2	81,920	204,800	40,959
111	ship_003	121,728	4,103,881	3,659
112	shipsec1	140,874	3,977,139	5,237
113	shipsec5	179,860	5,146,478	3,923
114	Si34H36	97,569	2,626,974	18,908
115	SiO2	155,331	5,719,417	55,068
116	sparsine	50,000	799,494	45,454
117	stokes128	49,666	295,938	33,282
118	stomach	213,360	3,021,648	20,021
119	t3dh	79,171	2,215,638	5,854
120	t3dh_a	79,171	2,215,638	5,854
121	thermal1	82,654	328,556	80,916
122	thermal2	1,228,045	4,904,179	1,226,000
123	torso3	259,156	4,429,042	212,262
124	TSOPF_FS_b162_c4	40,798	1,204,322	40,773
125	TSOPF_FS_b39_c19	76,216	998,359	76,206
126	turon_m	189,924	912,345	185,352
127	vanbody	47,072	1,191,985	3,526
128	venkat25	62,424	1,717,792	60,323
129	venturiLevel3	4,026,819	8,054,237	3,898,002
130	vibrobox	12,328	177,578	12,162
131	xenon1	48,600	1,181,120	10,001
132	xenon2	157,464	3,866,688	17,777

Table 4: Simulation results of test sparse matrices, all times are in milliseconds (1/3).

Mat. No	K_{MC60}	T_{MC60}	K_{SEQ}	T_{SEQ}	K_{HYB}	T_{HYB}
1	4,661	74	4,786	64.851	4,722	22.903
2	1,535	30	1,643	46.973	1,635	22.827
3	154	4	154	11.257	153	7.503
4	17,940	2667	18,376	2002.27	18,335	702.487
5	250	14	250	31.932	249	13.292
6	24,995	12	19,952	18.364	19,954	7.851
7	24,998	10	20,002	13.085	20,004	6.89
8	185	7	189	14.221	182	8.129
9	1,279	291	859	432.703	859	180.969
10	326	10	304	25.623	304	13.006
11	603	16	603	32.834	603	14.995
12	15,306	55	15,649	41.645	17,668	14.412
13	791	16	791	20.561	791	9.848
14	2,995	229	2,991	261.566	2,971	56.013
15	99,183	1209	99,106	69.376	99,106	24.253
16	18,985	41	16,294	40.311	16,295	15.885
17	7,182	409	7,182	714.248	7,182	221.985
18	35,948	2017	34,367	2250.05	34,742	995.987
19	26,317	74	26,352	90.209	25,671	42.155
20	1,683	14	2,290	10.883	1,699	7.013
21	17,433	30	18,840	25.899	20,108	9.386
22	1,217	26	1,176	32.168	1,185	12.114
23	720	38	555	47.609	555	24.221
24	59,998	17	40,012	21.205	40,012	10.468
25	5,334	236	5,330	323.123	5,331	122.311
26	3,290	202	3,294	281.516	3,271	120.057
27	87,569	1768	88,278	45.574	89,162	15.115
28	937	8	937	6.576	944	5.722
29	17,855	20	17,684	20.013	17,477	9.702
30	15,437	17	15,540	15.182	15,728	7.778
31	20,388	27	20,775	22.578	20,331	10.824
32	162,927	188	162,925	129.17	162,951	33.524
33	420,240	8812	417,648	10478.1	418,451	2965.8
34	508	56	275	90.555	275	44.651
35	20,109	20	20,157	33.552	20,072	14.904
36	20,109	20	20,157	26.421	20,056	14.918
37	2,938	47	3,040	46.074	3,041	22.55
38	2,371	71	2,498	77.337	2,119	33.949
39	36,386	25	39,594	22.137	35,823	10.376
40	9,239	17	10,256	20.655	9,005	10.352
41	16,290	65	16,908	15.907	16,718	8.253
42	404	15	403	17.13	404	8.484
43	604	35	603	40.053	604	14.629
44	2,332	28	2,306	30.285	2,425	12.139

Table 5: Simulation results of test sparse matrices, all times are in milliseconds (2/3).

Mat. No	K_{MC60}	T_{MC60}	K_{SEQ}	T_{SEQ}	K_{HYB}	T_{HYB}
45	3,038	218	3,197	284.313	3,087	133.367
46	34,706	741	34,060	860.578	33,545	295.679
47	2,577	53	2,576	75.252	2,579	20.905
48	745	33	969	26.857	804	14.099
49	23,493	9607	28,669	6488.37	29,434	1086.24
50	23,771	2161	23,329	2689.13	23,672	1208.47
51	6	8	7	11.082	7	7.348
52	1,003	37	1,012	34.034	1,023	15.89
53	1,599	24	1,830	23.684	1,835	12.436
54	286	16	284	25.386	283	12.888
55	443	1	546	1.784	461	3.144
56	847	20	807	56.834	814	27.264
57	127	5	139	14.515	139	9.044
58	4,326	97	4,357	164.92	4,460	68.775
59	3,491	67	3,413	103.496	3,637	34.581
60	1,300	21	1,320	24.249	1,319	10.42
61	5,068	485	5,078	494.318	5,069	91.82
62	11,117	24	10,692	34.876	10,622	16.973
63	12,272	135	12,502	173.601	12,645	73.338
64	10,821	83	10,757	104.777	10,646	40.982
65	361	7	359	20.224	359	12.711
66	1,939	44	1,722	48.557	1,873	21.059
67	4,206	168	4,322	214.259	4,322	99.196
68	402	18	404	16.04	404	9.052
69	17,741	1303	17,745	1237.56	17,742	332.087
70	7,220	51	7,119	58.132	7,320	27.416
71	5,041	140	5,528	32.403	5,725	13.79
72	38,747	2895	30,198	38.5	32,830	15.519
73	3,592	5020	3,351	3667.29	3,351	321.54
74	6,962	28	5,669	60.17	6,962	25.07
75	6,008	890	6,002	1082.91	6,040	467.64
76	4,859	16	4,959	19.354	5,036	9.382
77	1,683	13	2,290	10.781	1,699	6.972
78	1,598	20	1,200	25.359	1,632	9.362
79	2,857	85	2,651	180.566	2,483	89.264
80	9,162	1350	9,118	1402.45	9,092	564.026
81	4,343	10	5,638	12.327	5,611	8.468
82	8,794	63	8,849	89.498	9,052	33.994
83	2,820	54	2,820	69.44	2,820	22.279
84	11,596	16	11,593	29.965	11,594	12.73
85	1,179	113	1,187	110.15	1,178	30.345
86	2,176	18	2,197	20.594	2,212	9.542
87	2,186	20	2,175	28.364	2,212	13.485
88	4,046	49	4,046	106.271	4,047	47.231

Table 6: Simulation results of test sparse matrices, all times are in milliseconds (3/3).

Mat. No	K_{MC60}	T_{MC60}	K_{SEQ}	T_{SEQ}	K_{HYB}	T_{HYB}
89	202	6	202	7.649	202	5.773
90	1,682	23	1,698	15.545	1,682	8.268
91	10,699	544	10,620	784.474	10,618	378.682
92	861	10	603	32.872	603	15.008
93	7,828	4536	7,836	3277.96	7,995	415.608
94	19,554	274	21,031	205.045	21,135	57.146
95	2,078	70	2,176	86.528	2,155	38.611
96	1,215	32	1,208	32.341	1,209	11.307
97	2,028	18	2,026	21.378	2,075	9.123
98	514	121	513	148.497	514	48.255
99	2,513	64	2,609	111.419	2,669	50.118
100	9,373	90	9,271	208.533	9,224	79.866
101	2,034	249	2,040	274.646	2,036	120.387
102	1,845	40	1,048	38.839	1,048	19.932
103	1,845	40	1,048	38.482	1,048	20.907
104	423	23	421	25.344	423	10.371
105	7,850	50	9,374	17.956	9,831	8.896
106	618	41	603	130.431	603	54.741
107	6,827	10736	6,726	7580.49	6,725	692.11
108	1,171	95	614	110.344	614	52.106
109	322	12	322	15.355	322	7.786
110	322	12	322	14.39	322	7.779
111	3,863	181	3,659	188.254	3,659	89.757
112	5,968	157	5,237	193.02	5,237	84.959
113	3,988	160	3,923	242.555	3,923	109.121
114	16,275	124	15,621	157.212	15,856	63.661
115	20,466	312	20,842	346.286	20,570	137.269
116	22,560	67	25,660	53.534	22,346	26.269
117	756	18	768	17.074	769	9.392
118	1,143	82	1,125	144.645	1,132	67.573
119	5,476	87	5,088	122.055	4,927	51.945
120	5,476	88	5,088	123.657	5,854	53.024
121	215	26	481	22.633	225	10.92
122	797	675	1,849	425.648	885	111.231
123	7,881	152	6,957	235.145	6,954	98.409
124	20,269	53	20,445	56.03	20,208	30.181
125	38,235	38	38,291	51.238	38,100	24.233
126	3,425	63	2,988	74.329	2,956	21.447
127	2,012	51	2,007	54.619	1,976	27.97
128	1,495	44	1,515	102.236	1,515	39.35
129	1,741	1189	2,025	1211.39	2,025	169.062
130	4,356	8	4,601	11.562	4,497	7.802
131	1,932	32	2,115	62.915	2,033	28.585
132	3,765	102	3,689	206.947	3,686	86.418