TR-2014-01

An implementation of a reordering approach for increasing the product of diagonal entries in a sparse matrix

Ang Li, Radu Serban, Dan Negrut

May 24, 2015

Abstract

We present implementation details of a reordering strategy for permuting elements whose absolute value is large to the diagonal of a sparse matrix. This algorithm, based on work by Duff and Koster [9], is a critical component of the SPIKE-based preconditioner provided by the `SaP::GPU` library [2]. We discuss the four stages required to implement the equivalent bipartite graph matching problem and compare our implementation against the `MC64` algorithm provided by the HSL library [1]. The performance of the reordering algorithm is evaluated in terms of efficiency as well as the quality of the resulting `SaP::GPU` preconditioner. Numerical experiments, performed on more than 100 matrices arising in various engineering and scientific applications, indicate that our implementation is more efficient than `MC64` without any degradation in the quality of the resulting reordered matrix. The metric used in reaching this conclusion is the number of iterations required by the preconditioned sparse linear solver to approximate the solution within a prescribed tolerance.

Contents

1 Introduction

Row and/or column reordering operations that permute large (in absolute value) nonzero entries to the diagonal of a sparse matrix are useful in a variety of applications. In particular, such reorderings are a crucial element for the successful application of SPIKE-based methodologies as general preconditioners for iterative linear solvers [2]. Combined with subsequent reorderings aiming at bandwidth reduction, e.g., the reverse Cuthill-McKee (RCM) algorithm, these permutations lead to banded matrices with low bandwidth and high degree of diagonal dominance[1]. These properties are beneficial as they result in more effective preconditioned iterative linear solver by

- increasing the quality of the `SaP::GPU` preconditioner [14, 15],
- decreasing the probability of encountering a zero pivot during the factorization of diagonal blocks, and
- reducing the width of the *spike* blocks.

This report documents details of our implementation of one of the variants of the algorithm proposed by Duff and Koster [9]. We compare this implementation against the Harwell Standard Library (HSL) [1] algorithm `MC64`, both in terms of efficiency and in terms of the quality of resulting SPIKE-based preconditioner. For the latter, the metric used is the number of Krylov iterations required for converence by the resulting `SaP::GPU` preconditioner. Numerical experiments conducted on a series of over 100 different sparse matrices indicate that the proposed implementation runs faster than `MC64` while resulting in a comparable or even better preconditioner.

The remainder of this report is organized as follows. In §2, we provide a brief overview of the Duff-Koster algorithm [9]. The four stages of our proposed implementation are described in detail in §3. Numerical experiments and results of the comparisons against `MC64` are presented in §4. In §5 we discuss potential further improvements and various possibilities of parallelizing our implementation. Finally, §6 summarizes our findings.

2 Reordering algorithm description

The reordering algorithm is essentially a minimum bipartite perfect matching. Depending on which goal to be achieved (e.g. maximizing the absolute value of bottleneck, the sum, the product or other metrics of all diagonal entries), there are several variants of the algorithm. In our implementation, we select to maximize the product of all diagonal entries. This is to improve the overall diagonal dominances for the majority of rows. In this small section, we briefly introduce this variant of algorithm. Interested readers are refered to [9] for the details of other variants.

[1] 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 matrix $\{a_{ij}\}_{n\times n}$, the problem is to find a permutation σ s.t. $\prod_{i=1}^{n} |a_{i\sigma_i}|$ is maximized. Denote $a_i = \max_j |a_{ij}|$, note that a_i is an invariant of σ, then we are to minimize

$$\log \prod_{i=1}^{n} \frac{a_i}{|a_{i\sigma_i}|} = \sum_{i=1}^{n} \log \frac{a_i}{|a_{i\sigma_i}|} = \sum_{i=1}^{n} (\log a_i - \log |a_{i\sigma_i}|)$$

The reordering problem is reduced to minimum bipartite perfect matching in the following way: given a bipartite graph $G_C = (V_R, V_C, E)$, the weight c_{ij} of edge between node $i \in V_R$ and node $j \in V_C$ is defined as

$$c_{ij} = \begin{cases} \log a_i - \log |a_{ij}| & (a_{ij} \neq 0) \\ \infty & (a_{ij} = 0) \end{cases}$$

if we are able to find a minimum bipartite perfect matching σ s.t. $\sum c_{i\sigma_i}$ is minimized, according to the process of reduction above, $\prod_{i=1}^{n} |a_{i\sigma_i}|$ is maximized, which is exactly the goal of the reordering algorithm.

Numerous studies have been done on minimum bipartite perfect matching [3–5, 7, 11, 12]. A key concept in these works is the shortest augmenting path P. An augmenting path P for a match M starting at a certain row node i (or column node j) is said the be shortest if for any augmenting path P' starting at row node i (or column node j) for M, the weight of $M \oplus P$ is no greater than that of $M \oplus P'$[2]. Further more, works [12] and [10] stated that if M has minimum weight if and only if there exists dual variables u_i for $i \in V_R$ and v_j for $j \in V_C$ s.t.

$$\begin{cases} u_i + v_j \leq c_{ij} & ((i,j) \in E) \\ u_i + v_j = c_{ij} & ((i,j) \in M) \end{cases}$$

Define the reduced weight of edge (i, j) as

$$\overline{c}_{ij} = c_{ij} - u_i - v_j$$

With the observation that for any "valid" match M should satisfy $\overline{c}(M) = \sum_{(i,j)\in M} \overline{c}_{ij} = 0$, we can reach the conclusion that for any augmenting path P in the reduced graph $G_{\overline{C}}$, $\overline{l}(P) = c(P \backslash M) \geq 0$. Therefore, finding the shortest augmenting path in G_C is equivalent to finding the shortest path in $G_{\overline{C}}$, whose edge weights are all non-negative. In §3.3, the detailed implementation of finding the shortest path is discussed.

3 Implementation details

Our implementation of the algorithm described in §2 is separated into four stages:

[2]The alternating length of path P is defined as $l(P) := c(M \oplus P) = c(P \backslash M) - c(M \cap P)$, where $c(M)$ is the weight of matching M

Stage 1 – Form the bipartite graph,

Stage 2 – Find an initial partial match,

Stage 3 – Find a perfect match based on the initial (partial) match,

Stage 4 – Extract the permutation and scale arrays.

Note that while Stage 2 is technically optional, it acts as a relatively inexpensive warm start for Stage 3 and considerably decreases the overall execution time. The above four stages are described in detail in the remainder of this section.

3.1 Stage 1: Forming the bipartite graph

As indicated in §2, the problem of maximizing the product of diagonal elements can be reduced to finding the minimum perfect match of a bipartite graph with edge weights defined as:

$$c_{ij} = \begin{cases} \log(\max_j |a_{ij}|) - \log(|a_{ij}|) & (a_{ij} \neq 0) \\ \infty & (a_{ij} = 0) \end{cases}$$

Taking into account that the original matrix is assumed to be sparse, only edges with finite values are stored. In other words, we maintain the sparsity pattern of the input matrix and only modify the values of its nonzero elements. As such, the work in this stage is trivial and involves: (1) calculating the maximum absolute values in the original matrix for each row, and (2) updating each value to form the weighted bipartite graph.

3.2 Stage 2: Finding an initial partial match

As mentioned previously, this stage is not mandatory but the availability of an initial (partial) match as a starting point for Stage 3 was found to considerably reduce the running time for the overall algorithm. In our implementation, we use the strategy of Carpaneto and Toth [4]. After calculating

$$u_i = \min_j c_{ij}$$

and

$$v_j = \min_i (c_{ij} - u_i)$$

we try to match as many pairs of nodes as possible. The matched nodes (i, j) should satisfy $u_i + v_j = c_{ij}$. With this, we are essentially finding augmenting paths with length one.

Carpaneto and Toth also suggest an additional step in generating the initial match. As described above, when attempting to match a column node j, we are searching for a row node i that satisfies

$$u_i + v_j = c_{ij}\,.$$

Assume that the row node i was already matched to column node j', meaning that we also have

$$u_i + v_{j'} = c_{ij'} \, .$$

Therefore, if we can now match column node j' with some other row node i', with $i' \neq i$ such that

$$u_{i'} + v_{j'} = c_{i'j'} \, ,$$

we can rematch i to j and i' to j'. This essentially finds augmenting paths of length three.

However, we have opted not to use this *follow-up* strategy in our implementation, for the following two reasons:

- Numerical experiments indicate that the *follow-up* strategy has only marginal benefits as sufficiently many matches can be obtained without it;
- The *follow-up* strategy reduces the potential for parallelism in Stage 2 as it introduces additional contention and data non-locality.

3.3 Stage 3: Finding a perfect match

As shown by Duff and Koster, finding matches in the graph G_C is equivalent to finding shortest paths in the graph $G_{\overline{C}}$ with reduced edge lengths. See [9] for a detailed proof.

As the graph $G_{\overline{C}}$ contains no alternating paths with negative paths, we use Dijkstra algorithm [8] to find a shortest augmenting path. The augmenting path we are seeking is an alternate path with the pattern

$$i_0 \to j_0 \to i_1 \to j_1 \to \ldots \to i_k \to j_k \, ,$$

in which i_0 and j_k are unmatched, while j_p is matched with i_{p+1} for all $p = 0, 1, \ldots, n-1$. After each iteration, an updated match is obtained by matching i_p with j_p for all $p = 0, 1, \ldots, n$.

Dijkstra's algorithm is applied to all nodes i that are unmatched in the initial partial match obtained in Stage 2, thus ensuring that all row nodes (and therefore all column nodes) are eventually matched. Note that the algorithm is correct in the sense that: (i) after each iteration, one more row node and one more column node are matched and (ii) after each iteration, the arrays $\mathbf{u}$ and $\mathbf{v}$ are updated such that edges in the reduced graph $G_{\overline{C}}$ continue to have non-negative weights.

We note that the theoretical complexity of this stage is $O(n \cdot (m+n) \cdot \log n)$, where n and m are the dimension and number of nonzeros in the input matrix, respectively. However, as the numerical results in §4 show, thanks to the preprocessing Stage 2, the running time for finding a perfect match is acceptable in all situations and Stage 3 is the bottleneck only for about half of the matrices tested.

3.4 Stage 4: Extracting the permutation and the scaling factors

This post-processing stage is trivial. The resulting permutation can be obtained directly from the resulting perfect match: if the row node i was matched to the column node j then rows (or columns) i and j must be permuted.

Optionally, scaling factors can be calculated and applied to rows and columns in order to bring the matrix to a so-called I-matrix form; i.e., a matrix with 1 or -1 on the diagonal and off-diagonal elements of absolute value less than 1, see Olschowka and Neumaier [13]. This operation is highly parallelizable.

4 Evaluation

Performance of the proposed implementation was evaluated on a set of 116 sparse matrices from various applications. Most of these matrices were selected from the Florida Sparse Matrix Collection [6]. In addition, matrices ANCF31770 and ANCF88950 arise in the implicit integration of large-scale flexible body dynamics [16].

The reordering algorithm described in the previous sections was implemented in `SaP::GPU`, a library of preconditioned iterative linear solvers using a SPIKE-based preconditioner [2]. For evaluation purposes, we instrumented `SaP::GPU` to alternatively use the proposed and the HSL `MC64` implementations.

The comparison between proposed and the HSL implementations was based on the following metrics:

- Obtained value for the product of diagonal elements for the reordered matrix without scaling; to prevent floating point overflow, we report the logarithm of the absolute value of the product of diagonal elements, that is $\log|\prod_i a'_{ii}|$;
- Running time required to reorder, with separate times reported for each of the four individual stages;
- Quality of the resulting `SaP::GPU` preconditioner, defined as the number of BiCGStab(2) iterations required for solving $\mathbf{Ax} = \mathbf{b}$ using the resulting `SaP::GPU` preconditioner.

Preconditioner quality comparison. Results comparing the performance of the `SaP::GPU` solver using our implementation and the `MC64` algorithm are presented in Tables 1, 2 and 3. In these tables, `OM` indicates an out-of-memory `SaP::GPU` failure and `NC` indicates divergence (with a maximum number of iterations set at 550).

These results indicate that:

- Our implementation produces exactly the same LP value as that obtained by `MC64`, which implies that our implementation is at least no worse than `MC64` in terms of efficiency.

- With one exception (qa8fk), all problems that can be solved using `MC64` can also be solved using our reordering algorithm.
- There are seven problems that fail using `MC64` but can be solved using our algorithm.
- From the 69 problems that can be solved using either `MC64` or our reordering algorithm, in 65 cases, using `MC64` results in a larger number of Krylov iterations (the exceptions are apache1, ASIC_100ks, hcircuit and Lin).

Performance comparison. Tables 4, 5 and 6 display the breakdown of execution time of our reordering implementation over the four stages identified in §3 as well as the comparison against the running time for the `MC64` algorithm. All execution times are reported in milliseconds. Timing data for the four stages of our implementation is also displayed in Fig. 1, where the matrices are sorted in increasing order of the total time. Figure 2 shows the speedup obtained with our implementation over the HSL `MC64` algorithm.

These results indicate that:

- Our implementation runs faster than `MC64` for 96 of the 116 tested problems.
- In our implementation, the bottleneck is either Stage 2 (finding the initial match) or Stage 3 (finding a perfect match), with an approximately equal split among these two situations.

On using the follow-up strategy in Stage 2. In §3.2 we mentioned that our implementation does not use the *follow-up* strategy. This decision is supported by results reported in Fig. 3, which shows the speedup of the combined stages 2 and 3 with (as in the Duff and Koster algorithm) and without (as in our implementation) the follow-up step. Results are presented for every tenth matrix in our test set. As these results indicate, with one exception, the inclusion of the follow-up step leads to efficiency degradation which can be explained heuristically as follows. If the follow-up ends up with successfully matching the unmatched column node j, it requires that there are at least two potential matches i and i' for the second column node j', both of which (i.e. $\overline{c}_{ij'}$ and $\overline{c}_{i'j'}$) result in a zero reduced weight. This implies the following equation has to hold:

$$v_{j'} = c_{ij'} - u_i = c_{i'j'} - u_{i'}$$

This is seldom the case when all $\mathbf{u}$'s and $\mathbf{v}$'s are already assigned and unchanged. Moreover, row node i' should not have been already matched to another column node j'', which is an even less likely situation. As such, the follow-up step is, most often, unsuccessful (i.e., in most case, an unmatched column node j is still left unmatched).

Performance impact of a sequential implementation of Stage 3 on the GPU. A typical use case for the `SaP::GPU` linear solvers is within a loop (e.g. during time integration) with the system matrix and right-hand side generated or already available on the GPU. To maximize performance, it is desirable to keep the entire solution sequence on the GPU with

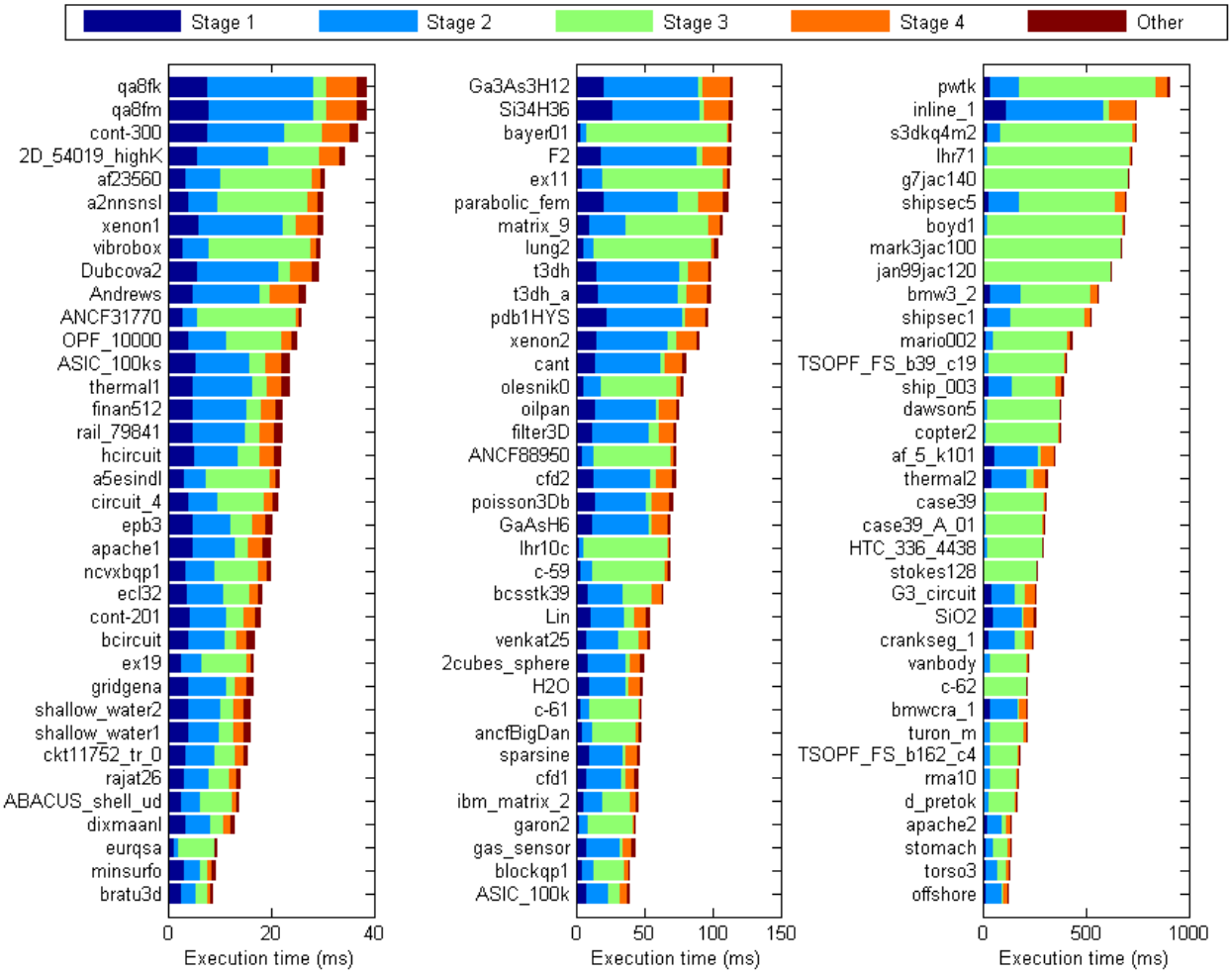

Figure 1: Breakdown of execution time over the four algorithmic stages.

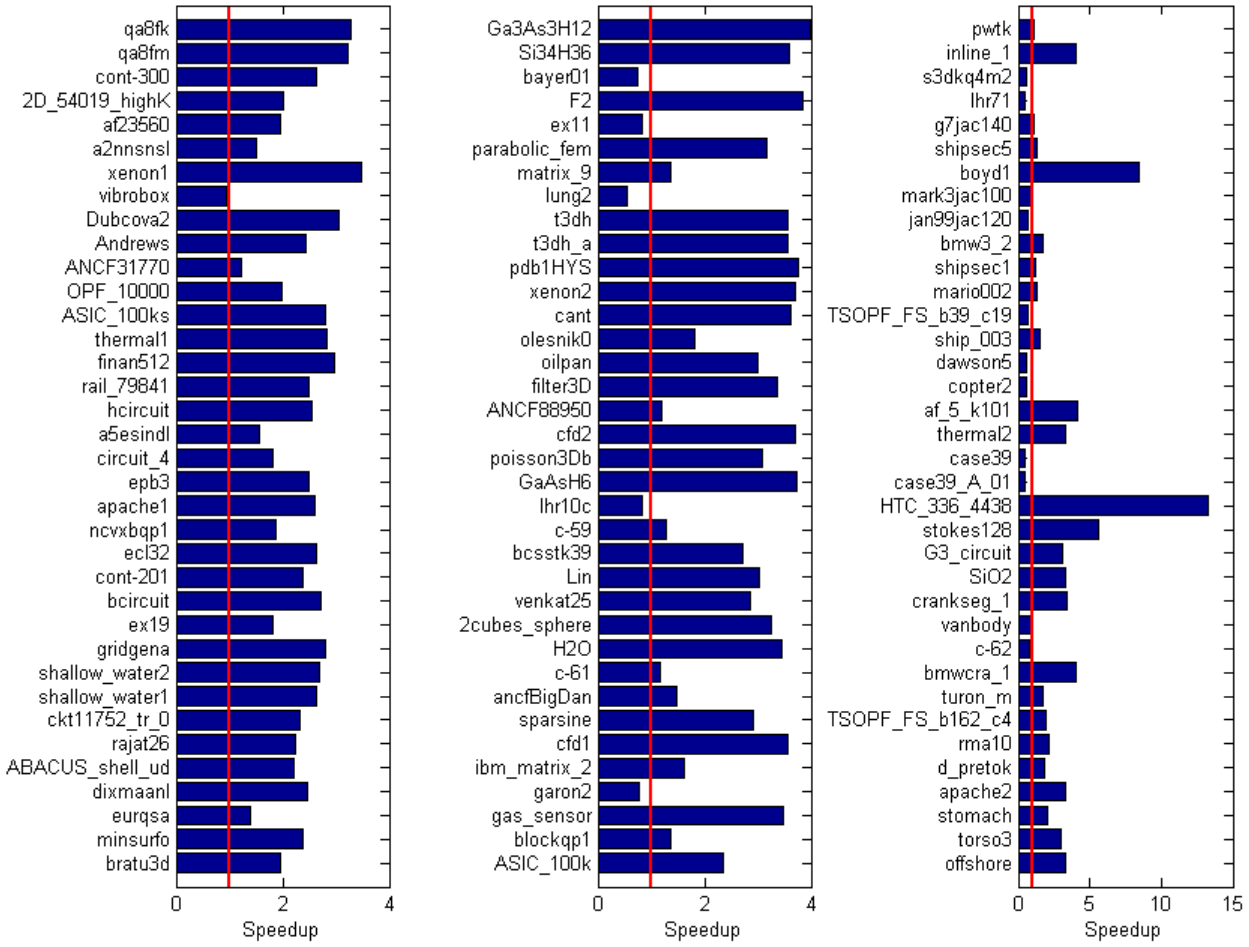

Figure 2: Speedup obtained with our implementation over the HSL `MC64` algorithm. A bar that extends beyond the red line indicates a test that runs faster than HSL `MC64`.

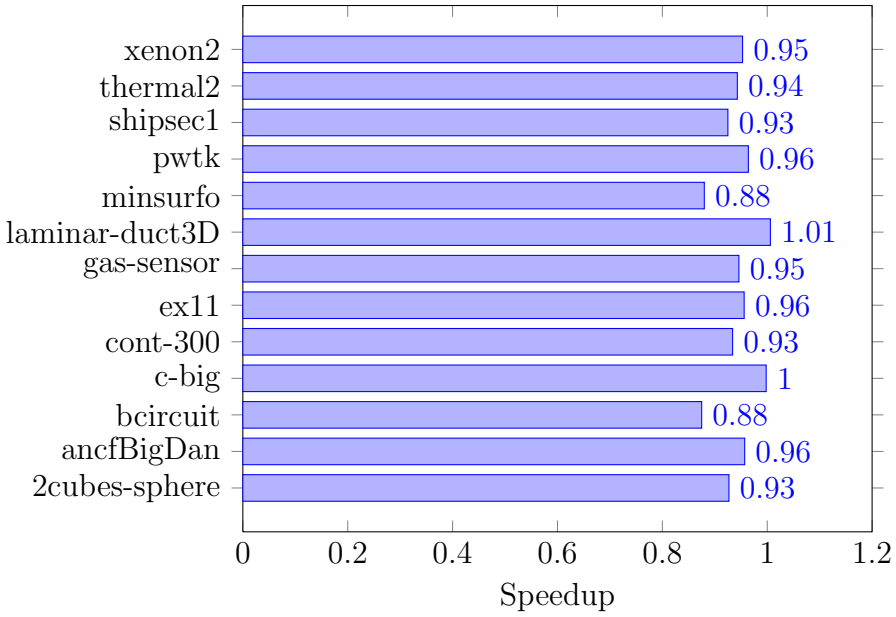

Figure 3: Speedup of combined stages 2 and 3 in the algorithm stated in Duff and Koster's work [9] compared to our implementation.

data transfer to the host (CPU) only when simulation results are required: e.g., for post-processing, visualization, etc. With the exception of the reordering algorithm discussed here, all other steps required for the SPIKE-preconditioned iterative linear system solution take place entirely on the GPU.

Since currently we are not aware of parallel algorithms that can implement Stage 3, it is worthwhile investigating the performance of a sequential implementation of the current Stage 3 on the GPU. The rationale behind this is the elimination of device-host-device data transfers and their impact on the overall efficiency. To this end, we launched a CUDA kernel with a single thread in a single block. Figure 4 summarizes the slowdown of the GPU-sequential implementation of Stage 3 compared to the original CPU-sequential implementation for all matrices in our test set. As these results show, the GPU-sequential implementation of Stage 3 is much slower for all matrices considered and, for almost half of them, it is two orders of magnitude slower. This suggests that this naive approach is impractical and, unless a properly GPU parallel implementation of Stage 3 can be obtained, the only option is to perform this stage on the CPU since even accounting for the data transfer the hybrid CPU/GPU implementation ends up winning.

5 Future work

Since Stage 1 and Stage 4 have already been fully parallelized on the GPU, we plan investigate how we can parallelize Stage 2 and possibly move it on the GPU as well. Note that a GPU

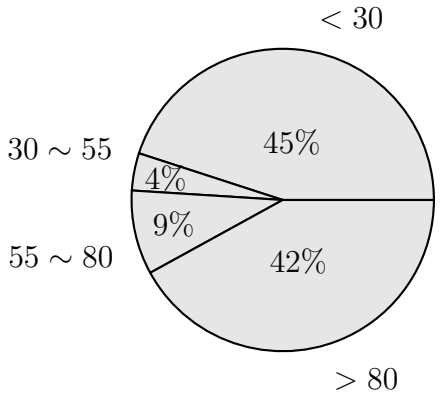

Figure 4: Efficiency comparison of a GPU-sequential vs. the CPU-sequential implementation of Stage 3 for all matrices in the test set. For instance, about 42% of the tests were more than 80 times slower, while 9% of the test were between 55 and 80 times slower.

solution is not necessarily mandatory. Currently, Stage 3 is implemented on the CPU and as such we could take advantage of the availability of the data on the CPU by implementing a multi-core solution for Stage 2. The biggest challenge here is contention, i.e. how we can tell whether different row nodes have selected the same column node, and when contention does happen, how we can resolve it? The more challenging task though remains the parallelization of Stage 3. We have no good lead at the present time. If one will be found, it is rather unlikely that a GPU implementation will prevail since this stage is loaded with flow control statements, a scenario where the GPU does not excel.

6 Conclusions

This document discusses an algorithm that seeks to maximize the product of the diagonal entries in a sparse matrix. The software implementation of the algorithm is compared in terms of speed and effectiveness with the HSL `MC64`. Two metrics are used to measure the effectiveness of the proposed and the `MC64` implementations: the size of the product of the diagonal entries in the permuted matrix and the effort required to solve a linear system whose coefficient matrix is reordered based on either the new or the `MC64` permutation. Note that, as part of the linear system solution sequence, a symmetric RCM step, identical for both approaches, is used to reduce the bandwidth after the diagonal boosting reordering. Our findings are as follows:

- The implementation proposed is faster than HSL `MC64` on a set of 96 out of 116 matrices, most of them chosen from the Florida sparse matrix collection. Speed-up values for the 116 tests are reported in Fig. 2.
- An attempt to move the entire solution onto the GPU turned out to slow the overall algorithm implementation tremendously. Specifically, the task in Stage 3, owing to the

sequential algorithm chosen to handle it, led to an implementation that was anywhere from 20 to 150 time slower than the CPU Stage 3 counterpart.

There are several caveats associated with the proposed implementation. First, it destroys the symmetry of the matrix it operates on. This is not specific to our implementation rather it is an attribute of the underlying diagonal boosting reordering algorithm. Second, it remains to determine whether a GPU-only or CPU-only implementation of the four-stage reordering algorithm would yield better outcomes in terms of efficiency when compared with the hybrid approach currently embraced in which Stage 1 and 4 are implemented on the GPU, while Stage 2 and 3 are carried out on the CPU. Data transfer overhead and the need to subsequently solve the linear system on the CPU or GPU are key factors that will determine the solution of choice: CPU only, GPU only, or hybrid.

References

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

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

[3] R. Burkhard and U. Derigs. *Assignment and matching problems: Solution methods with FORTRAN-programs*. Springer-Verlag New York, Inc., 1980.

[4] G. Carpaneto and P. Toth. Algorithm 548: Solution of the assignment problem [h]. *ACM Transactions on Mathematical Software (TOMS)*, 6(1):104–111, 1980.

[5] P. Carraresi and C. Sodini. An efficient algorithm for the bipartite matching problem. *European journal of operational research*, 23(1):86–93, 1986.

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

[7] U. Derigs and A. Metz. An efficient labeling technique for solving sparse assignment problems. *Computing*, 36(4):301–311, 1986.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[9] 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.

[10] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[11] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[12] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[13] M. Olschowka and A. Neumaier. A new pivoting strategy for gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[14] E. Polizzi and A. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006.

[15] E. Polizzi and A. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113 – 120, 2007.

[16] 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.

Table 1: Preconditioner quality comparison using our implementation vs. `MC64` (1/3).

Mat. No.	Mat. Name	Our LP	HSL LP	Our No. Iter.	HSL No. Iter.
1	2cubes_sphere	1.296×10^6	1.296×10^6	8.5	8.5
2	2D_54019_highK	-3.814×10^5	-3.814×10^5	0.25	NC
3	ANCF31770	-6.448×10^4	-6.448×10^4	8.25	7.75
4	ANCF88950	-1.799×10^5	-1.799×10^5	3.25	3.25
5	a2nnsnsl	-	-	OM	OM
6	a5esindl	-	-	OM	OM
7	ABACUS_shell_ud	1.004×10^5	1.004×10^5	0.25	NC
8	af_5_k101	6.651×10^6	6.651×10^6	0.25	0.25
9	af23560	8.748×10^4	8.748×10^4	7.75	8.25
10	ancfBigDan	-2.908×10^4	-2.908×10^4	2.25	2.25
11	Andrews	1.452×10^5	1.452×10^5	NC	NC
12	apache1	7.674×10^5	7.674×10^5	4.75	1.75
13	apache2	-	-	OM	OM
14	ASIC_100k	-6.036×10^5	-6.036×10^5	144.25	NC
15	ASIC_100ks	-6.036×10^5	-6.036×10^5	44.25	8.75
16	av41092	-9.573×10^3	-	NC	OM
17	bayer01	-5.628×10^5	-	5.75	OM
18	bcircuit	7.384×10^4	7.384×10^4	0.25	0.25
19	bcsstk39	8.268×10^5	8.268×10^5	0.25	0.25
20	blockqp1	1.823	1.823	7.75	8.25
21	bmw3_2	-	-	OM	OM
22	bmwcra_1	2.080×10^6	2.080×10^6	0.25	0.25
23	boyd1	5.494×10^5	5.494×10^5	349.5	444.25
24	bratu3d	3.799×10^4	3.799×10^4	52.75	52.75
25	c-59	-	-	OM	OM
26	c-61	-	-	OM	OM
27	c-62	-	-	OM	OM
28	c-big	-	-	OM	OM
29	cant	5.274×10^5	5.274×10^5	0.75	0.75
30	case39	-	-	OM	OM
31	case39_A_01	-	-	OM	OM
32	cfd1	0	0	0.25	0.25
33	cfd2	0	0	0.25	0.25
34	circuit_4	-	-	OM	OM
35	ckt11752_tr_0	3.390×10^5	3.390×10^5	0.25	2.25
36	cont-201	1.095×10^5	1.095×10^5	0.25	0.25
37	cont-300	2.474×10^5	2.474×10^5	1.75	1.25
38	copter2	-7.197×10^3	-7.197×10^3	1.75	1.75
39	crankseg_1	8.217×10^5	8.217×10^5	0.25	0.25
40	d_pretok	-2.794×10^4	-2.794×10^4	1.75	1.25

Table 2: Preconditioner quality comparison using our implementation vs. `MC64` (2/3).

Mat. No.	Mat. Name	Our LP	HSL LP	Our No. Iter.	HSL No. Iter.
41	dawson5	-4.720×10^3	-4.720×10^3	1.75	1.75
42	dixmaanl	2.935×10^5	2.935×10^5	0.25	0.25
43	Dubcova2	3.005×10^4	3.005×10^4	134.75	136.75
44	ecl32	-2.732×10^5	-2.732×10^5	246.25	225.25
45	epb3	-1.967×10^5	-1.967×10^5	0.25	0.25
46	eurqsa	4.335×10^4	4.335×10^4	NC	NC
47	ex11	2.250×10^5	2.250×10^5	NC	NC
48	ex19	4.511×10^4	4.511×10^4	254.25	NC
49	F2	9.851×10^5	9.851×10^5	NC	NC
50	filter3D	-7.010×10^5	-7.010×10^5	0.25	0.25
51	finan512	1.035×10^5	1.035×10^5	9.25	9.25
52	G3_circuit	-	-	OM	OM
53	g7jac140	-6.734×10^4	-6.734×10^4	NC	NC
54	Ga3As3H12	2.322×10^5	2.322×10^5	126.75	140.25
55	GaAsH6	-	-	OM	OM
56	garon2	6.593×10^3	6.593×10^3	0.25	0.25
57	gas_sensor	-4.893×10^5	-4.893×10^5	1.75	1.25
58	gearbox	-	-	OM	OM
59	gridgena	4.617×10^5	4.617×10^5	12.75	15.25
60	gsm_106857	-	-	OM	OM
61	H2O	3.081×10^5	3.081×10^5	159.25	148.25
62	hcircuit	-3.836×10^5	-3.836×10^5	372.25	222.25
63	HTC_336_4438	-	-	OM	OM
64	ibm_matrix_2	-9.144×10^5	-9.144×10^5	0.25	0.25
65	inline_1	-	-	OM	OM
66	jan99jac120	-1.809×10^4	-1.809×10^4	3.75	NC
67	laminar_duct3D	-2.107×10^5	-2.107×10^5	NC	NC
68	ldoor	-	-	OM	OM
69	lhr10c	-5.616×10^3	-5.616×10^3	1.75	NC
70	lhr71	-1.607×10^5	-	NC	OM
71	Lin	1.605×10^6	1.605×10^6	7.75	1.75
72	lung2	-1.224×10^6	-1.224×10^6	7.25	5.75
73	mario002	-3.201×10^4	-3.201×10^4	0.25	0.25
74	mark3jac100	-3.944×10^4	$-\infty$	0.75	NC
75	mark3jac140	-6.117×10^4	$-\infty$	NC	NC
76	matrix_9	-2.064×10^6	-2.064×10^6	252.25	254.25
77	minsurfo	1.772×10^4	1.772×10^4	44.75	44.75
78	ncvxbqp1	5.420×10^5	5.420×10^5	0.75	0.75
79	nd24k	-	-	OM	OM
80	offshore	-	-	OM	OM

Table 3: Preconditioner quality comparison using our implementation vs. `MC64` (3/3).

Mat. No.	Mat. Name	Our LP	HSL LP	Our No. Iter.	HSL No. Iter.
81	oilpan	7.238×10^5	7.238×10^5	0.25	0.25
82	olesnik0	3.122×10^4	3.122×10^4	9.25	8.75
83	OPF_10000	2.156×10^5	2.156×10^5	0.75	0.75
84	parabolic_fem	-4.839×10^5	-4.839×10^5	0.25	0.25
85	pdb1HYS	1.745×10^5	1.745×10^5	1.75	1.75
86	poisson3Db	-8.339×10^4	-8.339×10^4	165.25	167.5
87	pwtk	2.373×10^6	2.373×10^6	0.75	0.75
88	qa8fk	-9.913×10^4	-9.913×10^4	`NC`	199.25
89	qa8fm	-5.512×10^5	-5.512×10^5	11.25	11.25
90	rail_79841	-8.550×10^5	-8.550×10^5	0.25	0.25
91	rajat26	2.898×10^3	2.898×10^3	0.25	0.25
92	rma10	4.218×10^5	4.218×10^5	4.75	5.25
93	s3dkq4m2	5.211×10^4	5.211×10^4	0.25	0.25
94	shallow_water1	1.893×10^6	1.893×10^6	6.25	6.25
95	shallow_water2	1.958×10^6	1.958×10^6	14.25	14.25
96	ship_003	3.060×10^6	3.060×10^6	3.75	3.75
97	shipsec1	2.482×10^6	2.482×10^6	0.25	0.25
98	shipsec5	3.157×10^6	3.157×10^6	0.25	0.25
99	Si34H36	-	-	`OM`	`OM`
100	SiO2	-	-	`OM`	`OM`
101	sparsine	-	-	`OM`	`OM`
102	stokes128	-1.428×10^5	-1.428×10^5	`NC`	`NC`
103	stomach	-1.084×10^5	-1.084×10^5	61.5	63.25
104	t3dh	-5.905×10^5	-5.905×10^5	0.75	1.75
105	t3dh_a	-5.905×10^5	-5.905×10^5	0.75	1.75
106	thermal1	1.095×10^5	1.095×10^5	0.25	0.25
107	thermal2	-	-	`OM`	`OM`
108	torso3	-	-	`OM`	`OM`
109	TSOPF_FS_b162_c4	-	-	`OM`	`OM`
110	TSOPF_FS_b39_c19	-	-	`OM`	`OM`
111	turon_m	-2.593×10^5	-2.593×10^5	`NC`	`NC`
112	vanbody	6.093×10^5	6.093×10^5	`NC`	`NC`
113	venkat25	-3.575×10^5	-3.575×10^5	0.25	0.25
114	vibrobox	2.373×10^5	2.373×10^5	`NC`	`NC`
115	xenon1	3.155×10^6	3.155×10^6	0.25	0.25
116	xenon2	1.022×10^7	1.022×10^7	0.25	0.25

Table 4: Performance comparison between our implementation and `MC64` (1/3).

Mat. No.	T-first	T-second	T-third	T-fourth	T-other	T-total	T-HSL	Speedup
1	8.63	28.135	3.469	7.234	1.8857	49.3537	161	3.26217
2	5.846	13.671	10.042	3.882	0.7111	34.1521	69	2.02037
3	2.849	3.005	19.013	0.724	0.3175	25.9085	32	1.23512
4	5.033	7.919	56.199	2.654	1.2143	73.0193	88	1.20516
5	4.12	5.467	17.63	1.951	0.9581	30.1261	45	1.49372
6	3.356	4.139	12.246	1.362	0.5781	21.6811	34	1.56819
7	2.711	3.67	6.151	0.79	0.3008	13.6228	30	2.20219
8	60.573	209.574	15.945	62.431	2.966	351.489	1459	4.15091
9	3.578	6.748	17.677	1.793	0.4517	30.2477	59	1.95056
10	4.62	7.709	31.606	2.497	0.7574	47.1894	70	1.48338
11	5.012	12.847	2.105	5.476	1.1804	26.6204	65	2.44174
12	4.812	8.164	2.77	2.684	1.4295	19.8595	52	2.61839
13	26.73	65.218	21.017	23.482	2.758	139.205	459	3.2973
14	8.142	15.48	8.87	4.886	1.257	38.635	91	2.35538
15	5.489	10.357	3.25	3.067	1.4613	23.6243	66	2.79373
16	8.197	20.561	5418.54	6.173	0.719	5454.19	5563	1.01995
17	3.41	4.73	102.861	1.388	0.603	112.992	83	0.734565
18	4.146	6.924	2.331	2.001	1.2286	16.6306	45	2.70586
19	8.782	25.863	20.99	7.155	0.7552	63.5452	173	2.72247
20	5.318	7.952	22.372	2.792	0.7552	39.1892	53	1.35241
21	38.059	151.557	331.7	39.458	1.938	562.712	982	1.74512
22	35.474	135.402	5.309	36.422	1.791	214.398	869	4.05321
23	9.269	16.733	656.128	5.199	1.24	688.569	5849	8.49443
24	2.775	2.833	2.114	0.67	0.29872	8.69072	17	1.95611
25	4.009	7.825	53.982	1.995	0.6871	68.4981	87	1.27011
26	4.174	5.781	35.824	1.39	0.6252	47.7942	56	1.17169
27	4.29	9.291	199.937	2.266	0.717	216.501	166	0.76674
28	14.492	36.938	1684.3	12.806	1.574	1750.11	727	0.415402
29	14.518	48.237	2.284	13.564	1.3134	79.9164	289	3.61628
30	5.683	12.936	280.349	3.883	0.722	303.573	150	0.494115
31	5.672	13.1	276.63	3.81	0.732	299.944	150	0.500093
32	8.472	25.481	2.578	6.716	1.486	44.733	159	3.55442
33	12.925	41.936	4.127	11.582	1.777	72.347	268	3.70437
34	4.118	5.693	8.8	1.778	0.9329	21.3219	39	1.82911
35	3.539	5.566	4.103	1.573	0.6376	15.4186	36	2.33484
36	4.499	6.774	3.418	2.297	1.0109	17.9989	43	2.38903
37	7.875	14.808	7.258	5.409	1.347	36.697	97	2.64327
38	4.86	14.217	352.083	3.194	0.747	375.101	212	0.565181
39	33.793	124.539	48.999	35.237	1.359	243.927	836	3.42725
40	9.818	24.795	120.736	7.501	1.366	164.216	298	1.81468

Table 5: Performance comparison between our implementation and `MC64` (2/3).

Mat. No.	T-first	T-second	T-third	T-fourth	T-other	T-total	T-HSL	Speedup
41	5.577	17.13	352.784	3.78	0.749	380.02	232	0.610494
42	3.515	4.896	2.347	1.544	0.6398	12.9418	32	2.47261
43	5.798	15.665	2.34	4.098	1.1482	29.0492	89	3.06377
44	3.666	7.043	5.095	1.794	0.6633	18.2613	48	2.62851
45	5.017	7.309	4.259	2.406	1.0288	20.0198	50	2.49753
46	1.292	0.827	6.93	0.175	0.10915	9.33315	13	1.39288
47	5.302	14.475	88.022	3.676	0.663	112.138	93	0.829335
48	2.692	3.83	8.901	0.824	0.315	16.562	30	1.81138
49	18.653	70.51	3.579	18.735	1.39	112.867	435	3.85409
50	11.829	41.783	8.09	10.355	1.2457	73.3027	247	3.36959
51	4.895	10.537	2.621	2.803	1.3277	22.1837	66	2.97516
52	46.944	113.118	45.337	48.294	4.183	257.876	809	3.13717
53	4.413	8.235	692.677	2.307	0.704	708.336	755	1.06588
54	20.409	69.976	2.243	20.693	1.166	114.487	456	3.98298
55	12.604	41.098	2.24	11.827	1.1327	68.9017	257	3.72995
56	3.187	5.718	32.921	1.375	0.4133	43.6143	34	0.779561
57	8.077	24.435	2.477	6.277	1.4967	42.7627	149	3.48435
58	30.791	103.335	1302.21	31.019	1.825	1469.18	14407	9.80615
59	4.057	7.442	1.725	2.192	1.0368	16.4528	46	2.79588
60	73.919	308.538	567.888	124.387	2.818	1077.55	2032	1.88576
61	9.626	27.294	2.43	7.923	1.396	48.669	168	3.45189
62	5.332	8.439	4.122	2.77	1.222	21.885	56	2.55883
63	8.792	13.666	266.488	5.209	1.411	295.566	3917	13.2525
64	6.05	13.912	20.066	3.927	0.7504	44.7054	72	1.61054
65	117.48	471.18	25.521	129.044	2.942	746.167	3004	4.02591
66	3.384	4.316	612.694	1.196	0.587	622.177	421	0.676656
67	14.256	46.454	44296	13.033	0.657	44370.4	11823	0.266461
68	153.787	635.307	40.794	233.897	4.865	1068.65	3708	3.4698
69	2.578	3.394	61.744	0.695	0.2923	68.7033	58	0.84421
70	7.801	20.163	689.207	5.665	1.041	723.877	340	0.469693
71	11.249	24.079	7.806	8.814	2.0933	54.0413	164	3.03472
72	5.334	7.993	86.047	2.731	1.214	103.319	58	0.561368
73	14.211	40.56	356.949	16.577	1.585	429.882	567	1.31897
74	3.313	4.833	663.733	1.694	0.586	674.159	639	0.947848
75	3.939	6.358	1557.37	1.996	0.717	1570.38	1237	0.787707
76	10.475	26.318	60.73	8.116	1.262	106.901	147	1.3751
77	3.14	3.236	1.269	0.907	0.70656	9.25856	22	2.37618
78	3.553	5.678	8.301	1.679	0.6403	19.8513	37	1.86386
79	88.553	346.088	419.055	132.536	2.276	988.508	2263	2.28931
80	18.245	74.365	8.439	21.911	1.751	124.711	416	3.33571

Table 6: Performance comparison between our implementation and `MC64` (3/3).

Mat. No.	T-first	T-second	T-third	T-fourth	T-other	T-total	T-HSL	Speedup
81	14.175	44.445	2.692	12.269	1.4288	75.0098	226	3.01294
82	5.721	12.799	55.341	3.426	1.2139	78.5009	142	1.8089
83	3.986	7.522	10.603	2.044	0.6847	24.8397	49	1.97265
84	20.532	54.022	15.331	18.584	2.499	110.968	353	3.1811
85	22.52	56	1.9	14.513	1.1776	96.1106	361	3.75609
86	13.981	38.144	3.421	13.585	1.1556	70.2866	216	3.07313
87	38.893	142.42	661.129	56.945	2.34	901.727	974	1.08015
88	7.874	20.473	2.441	6.108	1.4626	38.3586	126	3.28479
89	7.889	20.374	2.442	6.118	1.5168	38.3398	124	3.23424
90	4.871	10.29	2.737	2.733	1.5037	22.1347	55	2.48479
91	3.273	4.694	4.093	1.235	0.5719	13.8669	31	2.23554
92	9.58	30.203	125.002	9.879	0.788	175.452	376	2.14304
93	25.168	62.158	639.011	16.391	1.701	744.429	405	0.544041
94	4.062	6.001	2.727	1.895	1.2663	15.9513	42	2.63301
95	4.103	6.041	2.71	1.89	1.2622	16.0062	43	2.68646
96	28.575	115.787	213.643	27.959	1.64	387.604	577	1.48863
97	27.763	111.082	354.967	31.386	1.817	527.015	623	1.18213
98	34.87	141.894	466.616	49.167	2.422	694.969	879	1.2648
99	26.629	64.04	3.693	18.101	1.892	114.355	411	3.59407
100	54.073	137.573	6.14	55.097	2.055	254.938	848	3.3263
101	9.586	24.545	3.065	8.461	0.806	46.463	135	2.90554
102	4.243	8.372	250.422	2.359	0.712	266.108	1488	5.59171
103	14.242	39.191	68.545	12.464	1.426	135.868	284	2.09026
104	15.888	60.03	6.237	15.125	1.2598	98.5398	351	3.56201
105	16.269	59.017	6.24	15.088	1.2684	97.8824	350	3.57572
106	4.908	11.567	2.801	2.931	1.3512	23.5582	67	2.84402
107	43.349	171.013	36.987	55.383	3.835	310.567	1017	3.27466
108	18.798	57.238	37.75	17.866	1.684	133.336	402	3.01494
109	10.241	27.338	135.263	8.222	0.732	181.796	361	1.98574
110	9.015	24.831	363.051	7.264	1.073	405.234	278	0.686023
111	10.132	25.786	167.571	7.933	1.389	212.811	371	1.74333
112	9.53	31.84	175.218	7.943	0.747	225.278	220	0.976571
113	7.826	23.599	15.192	6.286	0.7818	53.6848	154	2.8686
114	2.984	4.935	19.902	1.182	0.3903	29.3933	28	0.952598
115	6.079	16.219	2.638	4.265	0.7384	29.9394	104	3.47368
116	15.65	51.427	7.263	14.468	1.3597	90.1677	335	3.7153