Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU

M. Naumov, P. Castonguay and J. Cohen
NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050

Abstract

In this technical report we study different parallel graph coloring algorithms and their application to the incomplete-LU factorization. We implement graph coloring based on different heuristics and showcase their performance on the GPU. We also present a comprehensive comparison of level-scheduling and graph coloring approaches for the incomplete-LU factorization and triangular solve. We discuss their tradeoffs and differences from the mathematics and computer science prospective. Finally we present numerical experiments that showcase the performance of both algorithms. In particular, we show that incomplete-LU factorization based on graph coloring can achieve a speedup of almost 8× on the GPU over the reference MKL implementation on the CPU.

1 Introduction

The graph coloring algorithms have been studied by many authors in the past. The main objective of graph coloring is to assign a color to every node in a graph, such that no two neighbors have the same color and at the same time use as few colors as possible. Let us make this statement a bit more formal.

Let a graph $G(V, E)$ be defined by its vertex V and edge E sets. The vertex set $V = \{1, ..., n\}$ represents n nodes in a graph, with each node identified by a unique integer number $i \in V$. The edge set $E = \{(i_1, j_1), ..., (i_e, j_e)\}$ represents e edges in a graph, with each edge from node i to j identified by a unique integer pair $(i, j) \in E$.

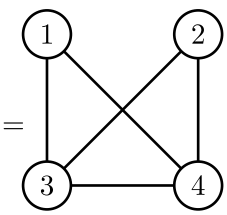
NVIDIA Technical Report NVR-2015-001, May 2015.

Also, let the adjacency matrix $A = [a_{i,j}]$ of a graph $G(V, E)$ be defined through its elements

$$a_{i,j} = \left\{ \begin{array}{ll} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{array} \right\} \tag{1}$$

Let us assume that if $(i, j) \in E$ then $(j, i) \in E$, in other words, the adjacency matrix is symmetric. If it is not, we can always work with $\bar{G}$ induced by $A + A^T$. An example of a graph G and its adjacency matrix A is shown below.

$$A = \begin{pmatrix} 0.0 & & 1.0 & 1.0 \\ & 0.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 0.0 \end{pmatrix}, \quad G(V, E) =$$



$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

Let us further define a function $f(i) : V \to C$, where $C = \{1, ..., k\}$ is a set of numbers, with each number representing a distinct color. Also, let $|C| = k$ be the number of colors, where $|C|$ denotes the cardinality (number of elements) of set C. In graph coloring we are interested in finding a function f that minimizes number of colors k, such that

$$\begin{array}{ll} \min\limits_{f} & |C| \\ \text{subject to} & f(i) \neq f(j) \text{ if } (i,j) \in E \end{array} \tag{2}$$

The achieved minimum is called the chromatic number $\chi(G)$ of a graph G.

The graph coloring problem stated in (2) for a general graph is NP-complete [20, 10]. However, there are many algorithms that can produce heuristic-based colorings that are a good enough approximation of the minimum coloring in a reasonable amount of time [19, 14].

This is especially true when graph coloring is applied to parallelize the incomplete-LU factorization preconditioner used in the iterative methods for the solution of large sparse linear systems. From the preconditioner perspective, fewer colors mean more parallelism, while more colors often imply stronger coupling to the original problem. Therefore, it usually does not hurt the preconditioner to have a few more colors than the theoretical minimum $\chi(G)$.

We will discuss applications to the incomplete-LU and sparse triangular solve in more detail later in the paper. Let us now focus on the coloring algorithms studied in great detail on the GPU in [9].

2 Graph Coloring

Let us first described a sequential algorithm that can perform approximate graph coloring. We will simply perform a breadth-first-search (BFS), and assign each node the smallest possible color among its neighbors, see Alg 1.

Algorithm 1 Sequential Graph Coloring

Let $G(V, E)$ be an input graph and S a set of root nodes.
Let C be an array of integers (representing colors), with
$C[r] = 1$ for $r \in S$, and
$C[r] = \infty$ otherwise.
while $S \neq \{\emptyset\}$ **do**
    Use some heuristic to order the vertices of S.
    for $v \in S$ **do**   ▷ Explore vertices in a given order
        Find the set of neighbors N and subset of visited neighbors W of v.
        C[v] $= \max_{w \in W} C[w] + 1$.
    end for
    Set $S = N \setminus W$.
end while

This is a special case of a greedy approach often used for the graph coloring problem [14]. It is not optimal, but it is simple to implement. A graph coloring obtained by this algorithm on a sample graph is shown in Fig 1.

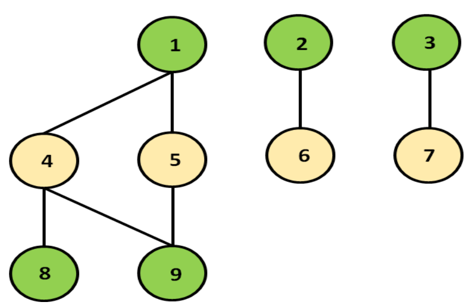


Figure 1: A sample graph coloring

In order to perform the coloring on a parallel platform, we could attempt to parallelize this algorithm. However, let us rather focus on a different more interesting approach that is based on the maximal independent set problem [21].

Let an *independent set* of graph $G(V, E)$ be a subset of vertices $W \subseteq V$, such that if $i, j \in W$ than $(i, j) \notin E$, in other words, no two vertices are adjacent.

Also, let a *maximal independent set* S be an independent set, such that $S \not\subset W$, for any other independent set W. Finally, let *maximum independent set* Z be a maximal independent set, such that $|Z| = \max_S |S|$, on other words, a maximal independent set with the largest cardinality.

Notice that a graph can have many maximal independent sets. For example, the graph on Fig. 1 has the following distinct maximal independent sets, with some of them being maximum independent sets:

$$\begin{array}{llll} \{4,5,6,7\}, & \{4,5,2,7\}, & \{4,5,2,3\}, & \{4,5,6,3\}, \\ \{8,5,6,7\}, & \{8,5,2,7\}, & \{8,5,2,3\}, & \{8,5,6,3\}, \end{array}$$
$$\{1,2,3,8,9\}, \{1,6,3,8,9\}, \{1,6,7,8,9\}, \{1,2,7,8,9\}, \leftarrow \text{maximum ind. sets}$$

Ideally we would like to find the maximum independent set, assign the same color to the nodes in it, and repeat. Unfortunately, this problem for a general graph is NP-complete. However, M. Luby developed a parallel algorithm for finding a maximal independent set [21], which can be used as an approximate solution to the original problem. His scheme is illustrated in Alg. 2 below.

Algorithm 2 Independent Set

```
1: Let G(V,E) be an input graph.
2: Let S = {∅} be the independent set.
3: Assign a pre-generated random number r(v) to each vertex v ∈ V.
4: for v ∈ V in parallel do                          ▷ Find local maximum
5:     if r(v) > r(w) for all neighbors w of v then
6:         Add vertex v to the independent set S.
7:     end if
8: end for
```

Algorithm 3 Graph Coloring

```
1: Let G(V,E) be the adjacency graph of the coefficient matrix A.
2: Let set of vertices W = V.
3: for k = 1,2,...until W = {∅} do ▷ Color an Independent Set Per Iteration
4:     Find in parallel an independent set S of W.
5:     Assign color k to vertices in S.
6:     Remove vertices in set S from W, so that W = W \ S
7: end for
```

Therefore, a common parallel approach for graph coloring is to leverage the parallel (maximal) independent set algorithm and implement coloring following the outline in Alg. 3, based on ideas by M. T. Jones and P. E. Plassman in [19].

Notice that it is possible to change the heuristics for selecting the nodes for the independent set on *line 3 - 5* of Alg. 2, therefore generating different and perhaps better approximations to the maximum independent set [2, 18, 1]. We illustrate a choice proposed by J. Cohen and P. Castonguay in [9], where:

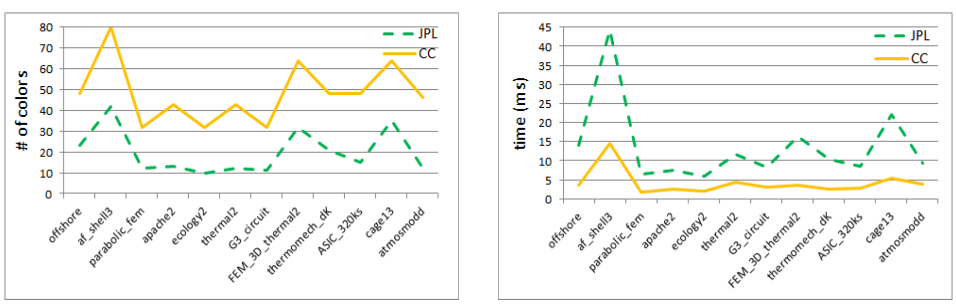
a) we use a hash function computed on-the-fly instead of random numbers.

b) we use maximum and minimum hash values to be able to generate two distinct (maximal) independent sets for each of the hash values.

c) we associate multiple hash values with each node, and use different hash values to create different pairs of (maximal) independent sets at once.

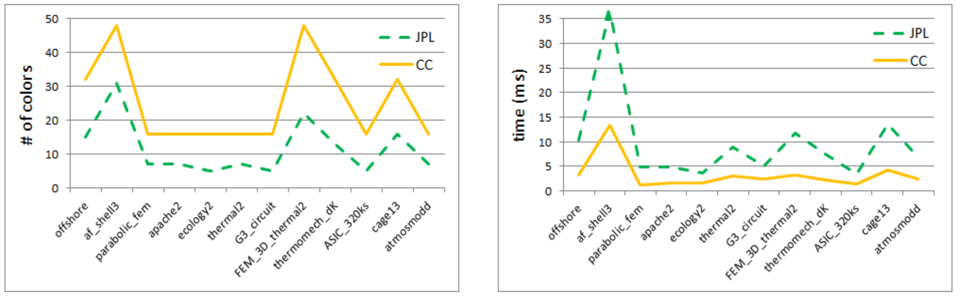
We compare Cohen-Castonguay (CC) with Jones-Plassman-Luby (JPL) approach described in Alg. 2 and 3 on realistic matrices from Tab. 2. We note that for the nonsymmetric matrices we work with the auxiliary $A + A^T$ matrix. We interpret these as adjacency matrices of some graph G according to (1). The JPL and CC algorithms are implemented in CUDA, with latter being available in the CUSPARSE library, through `csrcolor` routine. We perform the experiments using CUDA Toolkit 7.0 release on Ubuntu 14.04 LTS, with Intel 6-core i7-3930K 3.20 GHz CPU and Nvidia K40c GPU hardware.

Let us first focus on the case when the entire graph (100% of nodes) is colored with a given algorithm. The Fig. 2a and 2b show the number of colors needed for each graph and the time taken to compute them. Notice that the plots show that CC is roughly $3 - 4\times$ faster than the JPL algorithm. However, the CC algorithm also generates $2 - 3\times$ more colors. Therefore, in problems where the initial pre-processing time is not very important JPL is a reasonable algorithmic choice, and vice-versa for CC algorithm.



(a) # of colors (b) Time (ms)

Figure 2: JPL and CC algorithms for coloring 100% of graph nodes

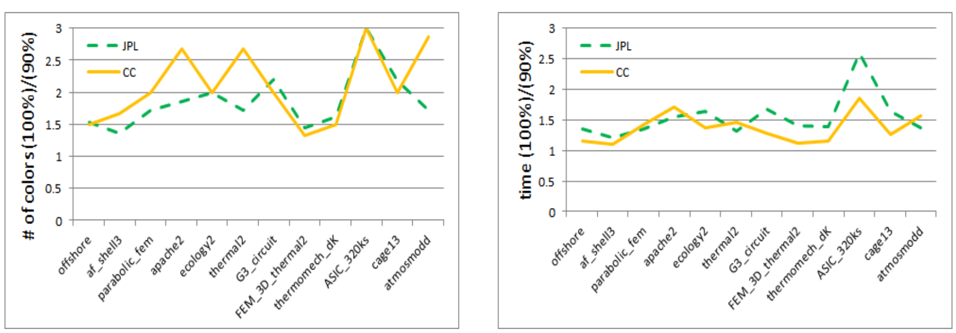


(a) # of colors (b) Time (ms)

Figure 3: JPL and CC algorithms for coloring 90% of graph nodes

Surprisingly in our numerical experiments roughly the same ratios hold when we color 80% and 90% of the graph nodes, with the latter case plotted in Fig. 3a and 3b. Also, notice that the number of necessary colors and computation time drops by $1.5 - 3.0\times$ and $1.2 - 2.4\times$, respectively, when only 90% of the graph nodes need to be colored, see Fig. 4a and 4b. Therefore, we can obtain a faster approximate solution if we have a scheme to process the rest of the nodes.

For example, if the nodes denote tasks and edges denote dependencies between them, then: (i) we can assign a distinct color for each of the remaining 10% of the nodes, so that the corresponding tasks are processed sequentially, or (ii) we can assign the same single color to the remaining 10% of the nodes, so that the corresponding tasks are processed in parallel (therefore ignoring the edge dependencies, if permitted by the underlying application).



(a) # of colors (b) Time (ms)

Figure 4: Ratio of JPL and CC algorithms between coloring 100% and 90% of graph nodes

Finally, there exist many re-coloring techniques that may improve an existing approximate coloring [18]. They however are beyond the scope of this paper and will not be discussed in greater detail here.

As the percentage of nodes to be colored is lowered further, for example to 80%, in our numerical experiment we see diminishing returns in terms of number of colors and required computation time. Also, in the case of incomplete-LU factorization, the set of remaining nodes becomes potentially too large to have its dependencies "ignored" even for preconditioning.

The detailed results are summarize in Tab. 1. It will be shown in later sections that in many cases the time needed for approximately coloring a graph corresponding to a given adjacency matrix is a small fraction of the time needed for solving the associated linear system using an iterative method with incomplete-LU preconditioning. We will explore this well known applications of graph coloring – incomplete-LU factorization – in the next section.

There are different variants of incomplete factorization that can benefit from reorderings based on graph coloring. We will focus on the incomplete-LU with 0 fill-in [ilu(0)]. The other variants, such as incomplete-LU with p-levels of fill-in [ilu(p)], are beyond the scope of this paper [27].

	JPL						CC					
	100%		90%		80%		100%		90%		80%	
mat rix	# col.	time (ms)	# col.	time (ms)	# col.	time (ms)	# col.	time (ms)	# col.	time (ms)	# col.	time (ms)
1.	23	14.2	15	10.3	13	9.59	48	3.72	32	3.20	32	3.29
2.	42	44.1	31	37.4	27	32.8	80	14.7	48	13.6	48	13.4
3.	12	6.99	7	4.63	6	4.12	32	1.92	16	1.26	16	1.25
4.	13	7.49	7	4.87	6	4.08	43	2.53	16	1.48	16	1.46
5.	10	5.75	5	3.51	5	3.52	32	2.06	16	1.50	16	1.51
6.	12	11.6	7	8.80	6	8.44	43	4.42	16	3.00	16	3.04
7.	11	8.02	5	5.20	4	4.19	32	2.97	16	2.31	16	2.31
8.	32	16.3	22	11.6	19	10.2	64	3.70	48	3.28	32	2.58
9.	21	10.4	13	7.56	12	7.48	48	2.46	32	2.12	32	2.18
10.	15	8.63	5	3.33	4	2.62	48	2.73	16	1.47	16	1.50
11.	35	22.0	16	13.4	14	11.6	64	5.45	32	4.32	32	4.28
12.	12	9.24	7	6.77	6	5.75	46	3.80	16	2.43	16	2.41

Table 1: # of colors and time used for coloring different % of nodes

3 Incomplete-LU Factorization

The incomplete-LU factorization with 0 fill-in is one of the most popular black-box preconditioners for iterative methods and smoothers for algebraic multigrid. The algorithm performs Gaussian elimination without pivoting of the coefficient matrix $A = [a_{ij}]$ of the sparse linear system

$$A\mathbf{x} = \mathbf{f} \tag{3}$$

where $A \in \mathbb{R}^{n \times n}$, the solution $\mathbf{x} \in \mathbb{R}^n$ and right-hand-side $\mathbf{f} \in \mathbb{R}^n$. The algorithm computes the lower $L = [l_{ij}]$ and upper $U = [u_{ij}]$ triangular factors, such that

$$A \approx LU \tag{4}$$

and sparsity pattern of A and $L + U$ is the same, in other words, the algorithm drops all elements that are not part of the original sparsity pattern of A, so that

$$\begin{cases} l_{ij} = 0 \text{ if } i > j \text{ or } a_{ij} = 0 \\ u_{ij} = 0 \text{ if } i < j \text{ or } a_{ij} = 0 \end{cases} \tag{5}$$

There are two distinct approaches to expose parallelism in the incomplete-LU factorization with 0 fill-in: (i) level-scheduling and (ii) graph coloring. Let us first explore both of them from the theoretical prospective, focusing on their distinct characteristics and tradeoffs. To illustrate the algorithms, let us consider the following symmetric coefficient matrix

$$A = \begin{pmatrix} a_{11} & & & a_{14} & a_{51} & & & & \\ & a_{22} & & & & a_{26} & & & \\ & & a_{33} & & & & a_{37} & & \\ a_{41} & & & a_{44} & & & & a_{48} & a_{49} \\ a_{51} & & & & a_{55} & & & a_{59} & \\ & a_{62} & & & & a_{66} & & & \\ & & a_{73} & & & & a_{77} & & \\ & & & a_{84} & & & & a_{88} & \\ & & & a_{94} & a_{95} & & & & a_{99} \end{pmatrix} \tag{6}$$

The first level-scheduling approach involves an implicit reordering of the linear system (3). In this approach we factor the original system, by finding which rows are independent, grouping them into levels, and processing all the rows within a single level in parallel [16, 24]. In this setting the "levels"' represent the data dependencies between groups of rows. Therefore, the next level can be processed only when the previous level has finished.

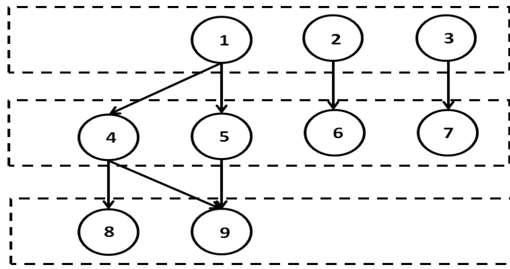


Figure 5: The data dependency DAG of the original matrix A

The directed acyclic graph (DAG) illustrating the data dependencies in the incomplete-LU factorization of the matrix in (6) is shown in Fig. 5. Note that in practice we do not need to construct the data dependency DAG because it is implicit in the structure of the matrix. There is a dependency between node i and j, for $i > j$ if there exists an element $a_{ij} \neq 0$ in the matrix.

The *analysis phase* of the level-scheduling scheme discovers already available parallelism. Notice that in this algorithm the node's children are visited only if they have no data dependencies on the other nodes. The independent nodes are grouped into levels, which are shown with dashed lines in Fig. 5. This information is passed to the *numerical factorization* phase, which can process the nodes belonging to the same level in parallel. Finally, an outline of the scheme is shown in Alg. 4 and 5.

Algorithm 4 Symbolic Analysis Phase

Let n and e be the matrix size and level number, respectively.
$e \leftarrow 1$
repeat ▷ Traverse the Matrix and Find the Levels
    for $i \leftarrow 1, n$ **do** ▷ Find Root Nodes
        if i has no data dependencies **then**
            Add node i to the list of root nodes.
        end if
    end for
    for $i \in$ the list of root nodes **do** ▷ Process Root Nodes
        Add node i to the list of nodes on level e.
        Remove the data dependency on i from all other nodes.
    end for
    $e \leftarrow e + 1$
until all nodes have been processed.

Algorithm 5 Numerical Factorization Phase

```
Let k be the number of levels.
for e ← 1, k do
    list ← the sorted list of rows in level e.
    for row ∈ list in parallel do              ▷ Process a Single Level
        Update elements in the row.
    end for
    Synchronize threads.                       ▷ Synchronize between Levels
end for
```

Since level-scheduling scheme can be viewed as an implicit reordering, we are still solving (3) and therefore the rate of convergence of iterative methods will not be affected when the level-scheduling scheme is used. It is important to note that physically shuffling the rows in memory, so that the rows belonging to the same level are next to each other from the computer science prospective, is irrelevant from the mathematical perspective because by construction the rows being shuffled are independent.

The second graph coloring approach to the incomplete-LU factorization [26, 7, 8, 6] involves an explicit reordering Q, so that we solve the reordered system

$$(Q^T AQ)(Q^T\mathbf{x}) = Q^T\mathbf{f} \tag{7}$$

The reordering Q is constructed based on graph coloring of the DAG illustrated in Fig. 5, that corresponds to the coefficient matrix in (6). The reordering results from the relabeling of nodes such that nodes of the same color are adjacent to one another. In our example, this relabeling permutation $\mathbf{q}^T = [1, 2, 3, 8, 9, 4, 5, 6, 7]$ corresponds to the reordering matrix

$$Q^T = \begin{pmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & 1 & & & & & & \\ & & & & & & & 1 & \\ & & & & & & & & 1 \\ & & & 1 & & & & & \\ & & & & 1 & & & & \\ & & & & & 1 & & & \\ & & & & & & 1 & & \end{pmatrix} \tag{8}$$

which results in the reordered coefficient matrix

$$
Q^T A Q = \left(\begin{array}{ccccc|cccc}
a_{11} & & & & & a_{14} & a_{15} & & \\
& a_{22} & & & & & & a_{26} & \\
& & a_{33} & & & & & & a_{37} \\
& & & a_{88} & & a_{84} & & & \\
& & & & a_{99} & a_{94} & a_{95} & & \\
\hline
a_{41} & & & a_{48} & a_{49} & a_{44} & & & \\
a_{51} & & & & a_{59} & & a_{55} & & \\
& a_{62} & & & & & & a_{66} & \\
& & a_{73} & & & & & & a_{77}
\end{array}\right) \tag{9}
$$

Notice that this matrix has diagonal blocks, which expose the parallelism available in the factorization and subsequent lower and upper triangular solves. The matrix updates during the factorization can now be performed trivially using diagonal scaling, matrix addition and multiplication.

Also, notice that if we look at the reordered matrix $Q^T AQ$ from the level-scheduling prospective, its data dependency DAG has wider and fewer levels as shown in Fig. 6. This implies that graph coloring extracts more parallelism than was originally available.

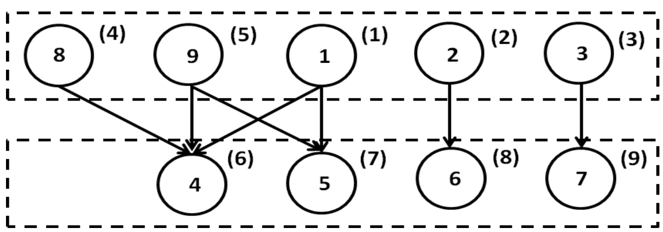


Figure 6: The data dependency DAG of the reordered matrix $Q^T AQ$

It is important to point out that physically shuffling the rows in memory, so that the rows belonging to the same color are next to each other from computer science prospective, is again irrelevant from the mathematical perspective because we have already changed the dependencies between them based on graph coloring. However, just as with level-scheduling, from the practical point of view such reshuffling can improve memory coalescing at the cost of some extra memory required to perform the shuffle.

Let us now address how the explicit reordering Q affects the convergence of an iterative method. The reordering Q is an orthogonal matrix ($Q^T Q = I$), therefore the reordered system (7) has been obtained from the original system

(3) using an orthogonal transformation. Since orthogonal transformation does not change the eigenvalues of a matrix, which govern the convergence of iterative methods, the unpreconditioned iterative methods are not affected [17].

However, when incomplete-LU preconditioning is added, the situation changes. It is more or less clear that depending on the reordering, different fill-in entries of the matrix will be dropped during the incomplete factorization, resulting in a preconditioner with a better or worse quality. Therefore, it follows that the convergence of the preconditioned iterative methods will be affected.

The authors are not aware of any general theoretical results about the effects of reordering on convergence, but there are many empirical studies, some of which are listed in these references [13, 12, 11, 5]. In our experiments, convergence was usually negatively affected by the graph coloring reordering, but the impact was not significant enough to offset gains obtained through the extra parallelism attained by coloring.

4 Sparse Triangular Solve

Finally, it is insightful to look at the difference between level-scheduling and graph coloring through the perspective of a standalone sparse triangular solve.

It has already been shown that the data dependency DAG for the incomplete-LU factorization in Fig. 5 and the corresponding lower triangular solve is the same [23, 24]. Therefore, a single level-scheduling analysis phase can be used to explore the available parallelism in both problems.

Let us now see what happens when we perform graph coloring on the lower triangular part of the matrix in (6) shown below

$$L = \begin{pmatrix} l_{11} & & & & & & & & \\ & l_{22} & & & & & & & \\ & & l_{33} & & & & & & \\ l_{41} & & & l_{44} & & & & & \\ l_{51} & & & & l_{55} & & & & \\ & l_{62} & & & & l_{66} & & & \\ & & l_{73} & & & & l_{77} & & \\ & & & l_{84} & & & & l_{88} & \\ & & & l_{94} & l_{95} & & & & l_{99} \end{pmatrix} \tag{10}$$

In order to perform the graph coloring of nonsymmetric matrix L, we work with $\bar{G}$ induced by $L + L^T$, which has the same sparsity pattern as A, and therefore results in the same reordering Q in (8). Then,

$$Q^T L Q = \left(\begin{array}{ccccc|cccc}
l_{11} & & & & & & & & \\
 & l_{22} & & & & & & & \\
 & & l_{33} & & & & & & \\
 & & & l_{88} & & {\color{red} l_{84}} & & & \\
 & & & & l_{99} & {\color{red} l_{94}} & {\color{red} l_{95}} & & \\
\hline
l_{41} & & & & & l_{44} & & & \\
l_{51} & & & & & & l_{55} & & \\
 & l_{62} & & & & & & l_{66} & \\
 & & l_{73} & & & & & & l_{77}
\end{array}\right) \tag{11}$$

Notice that the data dependency DAG of the $Q^T LQ$ is exactly the same as that of L, and that of A shown in Fig. 5. Therefore, graph coloring did not help us expose more parallelism in standalone sparse triangular solve.

Also, notice that the only way to attain the data dependency DAG of $Q^T AQ$ shown in Fig 6, is to symmetrically switch the three elements in red from the upper to the lower triangular part of the matrix

$$\left(\begin{array}{ccccc|cccc}
l_{11} & & & & & & & & \\
 & l_{22} & & & & & & & \\
 & & l_{33} & & & & & & \\
 & & & l_{88} & & & & & \\
 & & & & l_{99} & & & & \\
\hline
l_{41} & & & {\color{red} l_{84}} & {\color{red} l_{94}} & l_{44} & & & \\
l_{51} & & & & {\color{red} l_{95}} & & l_{55} & & \\
 & l_{62} & & & & & & l_{66} & \\
 & & l_{73} & & & & & & l_{77}
\end{array}\right) \tag{12}$$

Consequently, the only way to extract more parallelism is to break the existing data dependencies and create new ones. This is exactly what happens in the incomplete-LU factorization, where the explicit reordering based on graph coloring sets up a new problem with different data dependencies, that is better suited for parallelism, but results in a different preconditioner from the original problem. Notice that the switch of elements from lower to upper triangular part also happens in the case of incomplete-LU, but is harder to spot in (9) due to a symmetric sparsity pattern.

Let us now compare the level-scheduling and graph coloring approaches on a set of realistic matrices.

5 Numerical Experiments

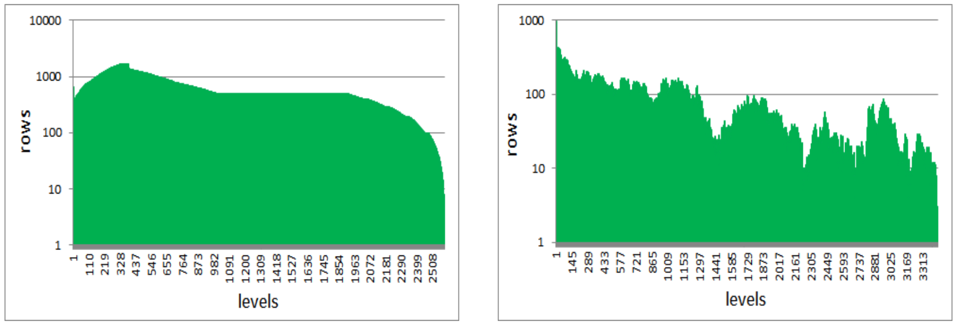
In this section we study the performance of the incomplete-LU factorization with 0 fill-in. In particular, we are interested in the speedup obtained by the numerical factorization phase using level-scheduling with and without a prior explicit reordering of the matrix. Notice that in the former case we will simply be using level-scheduling, while in the latter case we will use reordering resulting from graph coloring of the adjacency graph of the coefficient matrix.

We use twelve matrices selected from The University of Florida Sparse Matrix Collection [28] in our numerical experiments. The seven symmetric positive definite (s.p.d.) and five nonsymmetric matrices with the respective number of rows (m), columns (n=m) and non-zero elements (nnz) are grouped and shown according to their increasing order in Tab. 2.

#	Matrix	m,n	nnz	s.p.d.	Application
1.	offshore	259,789	4,242,673	yes	Geophysics
2.	af_shell3	504,855	17,562,051	yes	Mechanics
3.	parabolic_fem	525,825	3,674,625	yes	General
4.	apache2	715,176	4,817,870	yes	Mechanics
5.	ecology2	999,999	4,995,991	yes	Biology
6.	thermal2	1,228,045	8,580,313	yes	Thermal Simulation
7.	G3_circuit	1,585,478	7,660,826	yes	Circuit Simulation
8.	FEM_3D_thermal2	147,900	3,489,300	no	Mechanics
9.	thermomech_dK	204,316	2,846,228	no	Mechanics
10.	ASIC_320ks	321,671	1,316,085	no	Circuit Simulation
11.	cage13	445,315	7,479,343	no	Biology
12.	atmosmodd	1,270,432	8,814,880	no	Atmospheric Model.

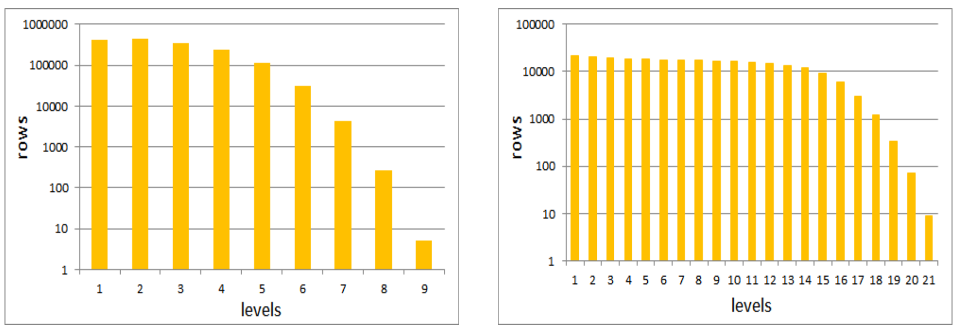
Table 2: Symmetric positive definite (s.p.d.) and nonsymmetric test matrices

The experiments are performed using `csrilu0` and `csrilu02` routines that implement different variants of level-scheduling (as well as the `GetLevelInfo` routine that returns extra information about distribution of rows into levels). The graph coloring is performed using `csrcolor` routine, that implements CC algorithm, for 100% of graph nodes. All of these routines are implemented on the GPU as part of the CUSPARSE library [25]. Also, we compare our performance to the reference `csrilu0` implementation on the CPU in Intel MKL [22]. The experiments are performed with CUDA Toolkit 7.0 release and MKL 11.0.4 on Ubuntu 14.04 LTS, with Intel 6-core i7-3930K 3.20 GHz CPU and Nvidia K40c GPU hardware.



(a) G3_circuit (b) offshore

Figure 7: Distribution of rows into levels for level-scheduling approach `csrilu0`



(a) G3_circuit (b) offshore

Figure 8: Distribution of rows into levels for graph coloring approach `csrilu0`

First, let us take a closer look at the distribution of rows into levels with and without prior graph coloring. We plot this distribution for G3_circuit and offshore matrices for level-scheduling (no prior reordering) in Fig. 7. In this case there are roughly up to 2000 and 1000 rows per level for these matrices, respectively. Then, the same plot is shown after graph coloring in Fig. 8. Notice that now we have more than $400,000$ and $20,000$ rows per level. Recall that rows in a single level can be processed in parallel, therefore the degree of available parallelism has increased more than an order of magnitude after graph coloring.

Let us also take a look at the difference in the number of levels, which represent data dependencies, that are required for each matrix. We plot it in Fig. 9. Notice that the difference varies across matrices, but in our numerical experiments it is not uncommon for it to be of two orders of magnitude.

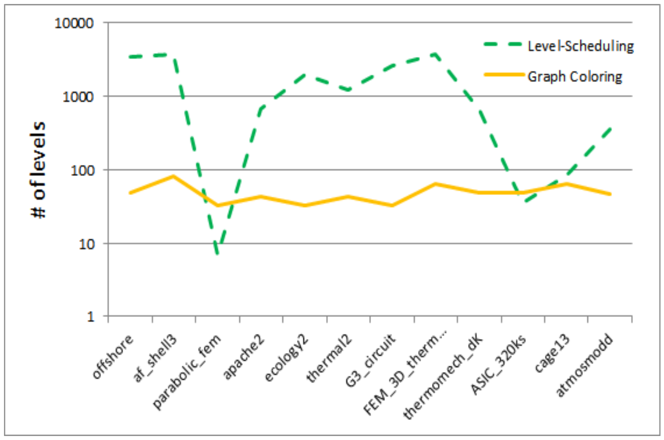


Figure 9: Number of levels resulting from level-scheduling and graph coloring reorderings

An interesting observation is that for parabolic_fem graph coloring resulted in a slightly worse distribution of rows into levels. This is very rare, but could happen because we are using approximate coloring algorithms. Fortunately, the decrease in degree of parallelism is small.

Finally, the speedup of the numerical factorization phase of the incomplete-LU factorization in CUSPARSE on GPU vs. MKL on CPU is shown in Fig. 10. Notice that the difference in performance follows the improvement in the degree of available parallelism, which mirrors the better distribution of rows into levels.

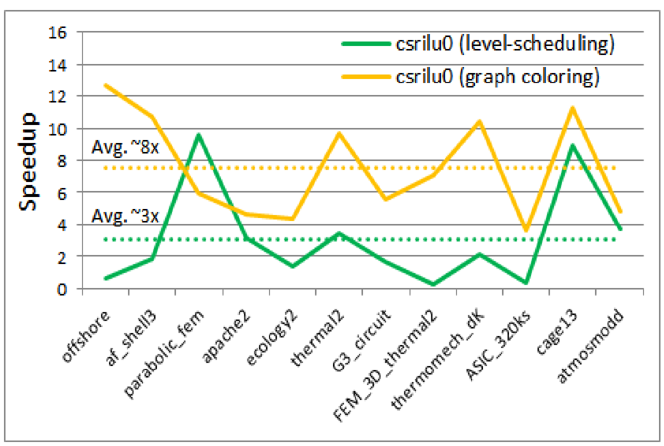


Figure 10: Speedup of numerical fact. based on graph coloring & level-scheduling vs. MKL

In our numerical experiments, on average numerical factorization phase of level-scheduling (with no prior explicit reordering) attains $3\times$ speedup, while using explicit reordering based on graph coloring allows us to reach almost $8\times$ speedup over the CPU. The detailed timing results are shown in Tab. 3. Notice that the time for the `csrcolor` routine is higher than shown in Tab. 1 for coloring 100% of nodes with CC algorithm, because here we explicitly generate the reordering vector with it. Also, note that additional speedup in `csrilu0` can often be obtained using JPL algorithm and recoloring techniques.

	CUSPARSE				MKL
	level-scheduling		graph coloring		
#	`csrilu0_analysis`	`csrilu0` (fact.)	`csrcolor` (coloring)	`csrilu0` (fact.)	`csrilu0` (fact.)
1.	79.39	410.4	5.15	20.14	255.3
2.	154.4	596.6	17.0	103.1	1102.
3.	36.50	6.210	4.64	9.970	59.50
4.	53.13	17.71	6.18	12.17	56.30
5.	74.81	35.09	7.21	10.86	47.22
6.	92.30	61.00	11.0	21.88	211.1
7.	114.4	57.18	11.2	16.66	92.50
8.	76.80	539.3	4.28	19.31	136.4
9.	34.79	54.74	3.55	11.38	118.7
10.	23.23	174.3	4.49	18.97	69.55
11.	53.89	44.32	7.73	35.34	397.0
12.	86.85	28.67	11.9	22.11	107.5

Table 3: Time(ms) for the incomplete-LU factorization

6 Conclusion

In this paper we have explored graph coloring algorithms and their application to the incomplete-LU factorization.

We noticed that exact graph coloring is the best reordering for extracting parallelism from a given problem. However this problem is NP-complete. Fortunately, there are many approximate graph coloring schemes. We presented one such novel algorithm (CC) and compared it with the standard approach (JPL). We noticed that the former approach, implemented in the CUSPARSE library, finds the solution faster, while the latter often has better quality.

We have also explained the relationship between level-scheduling and graph coloring approaches to the incomplete-LU factorization. In our numerical experiments we have shown that using graph coloring we can improve the performance of the numerical factorization phase of the incomplete-LU by almost $8\times$ when compared to the Intel MKL reference implementation on the CPU.

Finally, we note that the advantages of using graph coloring will vary greatly depending on the degradation of convergence of an iterative method. However, we believe that for many problems the additional degree of parallelism and the resulting speedup will often outweight this disadvantage.

7 Acknowledgements

The authors would like to acknowledge Joe Eaton and Michael Garland for their useful comments and suggestions.

References

[1] M. Adams, *A Parallel Maximal Independent Set Algorithm*, Proc. Copper Mountain, 1998.

[2] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer and C. L. Martin, *A Comparison of Parallel Graph Coloring Algorithms*, 1995.

[3] P. R. Amestoy, T. A. Davis and I. S. Duff *An Approximate Minimum Degree Ordering Algorithm*, SIAM Matrix Anal. Appl., pp. 886-905 (17), 1996.

[4] R. Barrett, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1994.

[5] M. Benzi, D. B. Szyld and A. van Duin, *Orderings for Incomplete Factorization Preconditioning of Nonsymmetric Problems*, SIAM J. Sci. Comput., pp. 1652-1670 (20), 1999.

[6] E. G. Boman, D. Bozdag, U. Catalyurek and A. H. Gebremedhin, *A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers*, Proc. Euro. Par. Proc., pp. 241-251, 2005.

[7] E. F. F. Botta and A. van der Ploeg, *Renumbering Strategies Based on Multilevel Techniques Combined with ILU Decompositions*, Zh. Vychisl. Mat. Mat. Fiz. (Comput. Math. Math. Phys.), pp. 1294-1300 (37), 1997.

[8] E. F. F. Botta and F. W. Wubs, *Matrix Renumbering ILU: An Effective Algebraic Multilevel ILU Preconditioner for Sparse Matrices*, SIAM J. Matrix Anal. Appl., Vol. 20, pp. 1007-1026, 1999.

[9] J. Cohen and P. Castonguay *Efficient Graph Matching and Coloring on the GPU*, GPU Tech., GTC on-demand S0332, 2012.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 2nd Ed., 2001.

[11] I. S. Duff and G. A. Meurant, *The Effect of Ordering on Preconditioned Conjugate Gradients*, BIT, pp. 635-657 (29), 1999.

[12] L. C. Dutto, *The Effect of Ordering on Preconditioned GMRES Algorithm for Solving the Compressible Navier-Stokes Equations*, Internat. J. Numer. Methods Eng., pp. 457-497 (36), 1993.

[13] H. C. Elman and E. Agron, *Ordering Techniques for the Preconditioned Conjugate Gradient Method on Parallel Computers*, Comput. Phys. Comm., pp. 253-269 (53), 1989.

[14] A. H. Gebremedhin, *Parallel Graph Coloring*, Ph.D. Thesis, University of Bergen, Norway, 1999.

[15] A. George and J. Liu *The evolution of the Minimum Degree Ordering Algorithm*, SIAM Review, pp. 1-19 (31), 1989.

[16] P. Gonzalez, J. C. Cabaleiro and T. F. Pena, *Parallel Incomplete LU Factorization as a Preconditioner for Krylov Subspace Methods*, Parallel Proc. Letters, pp. 467-474 (9), 1999.

[17] R. A. Horn and C. R. Johnson, *Matrix Analysis*, Cambridge University Press, New York, NY, 1999.

[18] T. R. Jensen and B. Toft, *Graph Coloring Problems*, John Wiley and Sons, New York, NY, 1995.

[19] M. T. Jones and P. E. Plassman, *A Parallel Graph Coloring Heuristic*, SIAM J. Sci. Comput., pp. 654-669 (14), 1992.

[20] R. M. Karp., *Reducibility among Combinatorial Problems*, Proc. Complexity of Computer Computations, pp. 85-103, 1972.

[21] M. Luby, *A Simple Parallel Algorithm for the Maximal Independent Set Problem*, Proc. ACM Symp. Theory Comput., pp. 1-10, 1985.

[22] "Intel Math Kernel Library",
http://software.intel.com/en-us/articles/intel-mkl

[23] M. Naumov, *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*, Nvidia Technical Report, 1, 2011.

[24] M. Naumov, *Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU*, Nvidia Technical Report, 3, 2012.

[25] Nvidia, *CUSPARSE and CUBLAS Libraries*,
http://developer.nvidia.com/cuda-downloads

[26] M. Pakzad, *J. L. Lloyd and C. Philipps, "Independent Columns: A New Parallel ILU Preconditioner for the PCG Methods*, Parallel Comput., pp. 583-605 (21), 1995.

[27] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2nd Ed., 2003.

[28] The University of Florida Sparse Matrix Collection,
http://www.cise.ufl.edu/research/sparse/matrices/

Appendix - Iterative Methods

This appendix was added to illustrate the difference in performance of preconditioned iterative methods using level-scheduling and graph coloring approaches. In particular, we experiment with Bi-Conjugate Gradient Stabilized (BiCGStab) and Conjugate Gradient (CG) iterative methods for nonsymmetric and s.p.d. systems, respectively. These methods are preconditioned with incomplete-LU in (4) and Cholesky $A \approx R^T R$ factorizations with 0 fill-in, respectively.

We compare their implementation using the CUSPARSE and CUBLAS libraries on the GPU and MKL on the CPU. In our experiments we let the initial guess be zero, the right-hand-side $\mathbf{f} = A\mathbf{e}$ where $\mathbf{e}^T = (1, \ldots, 1)^T$, and the stopping criteria be the maximum number of iterations 2000 or relative residual $||\mathbf{r}_i||_2/||\mathbf{r}_0||_2 < 10^{-7}$, where $\mathbf{r}_i = \mathbf{f} - A\mathbf{x}_i$ is the residual at i-th iteration. The experiments are performed with CUDA Toolkit 7.0 release and MKL 11.0.4 on Ubuntu 14.04 LTS, with Intel 6-core i7-3930K 3.20 GHz CPU and Nvidia K40c GPU hardware.

	CPU (reference)			GPU (level-scheduling)			GPU (graph coloring)		
#	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.
1.	0.45	8.83E-08	25	2.00	8.83E-08	25	10.67	†	2000
2.	24.4	9.74E-08	570	46.5	9.71E-08	570	12.16	9.99E-08	723
3.	22.3	9.85E-08	1044	3.55	9.83E-08	1044	6.36	9.95E-08	1106
4.	22.7	9.97E-08	713	10.7	9.97E-08	713	8.66	9.90E-08	1377
5.	75.5	9.98E-08	1746	65.4	9.98E-08	1746	17.23	9.96E-08	2447
6.	109.	9.99E-08	1655	48.1	9.90E-08	1655	15.44	9.99E-08	1348
7.	13.6	8.51E-08	183	9.51	8.22E-08	183	3.48	9.94E-08	300
8.	0.10	5.25E-08	4	0.74	5.25E-08	4	0.19	7.21E-08	10
9.	58.5	1.56E-04	2000	42.3	1.96E-04	2000	14.92	1.41E-04	2000
10.	0.15	6.33E-08	6	0.16	6.33E-08	6	0.18	9.09E-08	8
11.	0.17	2.52E-08	2.5	0.22	2.52E-08	2.5	0.24	5.51E-08	3
12.	7.95	8.19E-08	74.5	3.06	9.62E-08	75	2.57	8.64E-08	105

Table 4: `csrilu0` preconditioned CG and BiCGStab methods

The results of the numerical experiments are shown in Tab. 4, where we state the number of iterations required for convergence (# it.), achieved relative residual ($\frac{||\mathbf{r}_i||_2}{||\mathbf{r}_0||_2}$) and time in seconds taken by the iterative solution of the linear system (solve). Notice that the solve time excludes the graph coloring (`csrcolor`), level scheduling (`csrilu0_analysis`) and numerical factorization (`csrilu0`) time, that has already been shown in Tab 3. Also, note that here the solve time is in seconds (s), while in previous tables it is in milliseconds (ms).

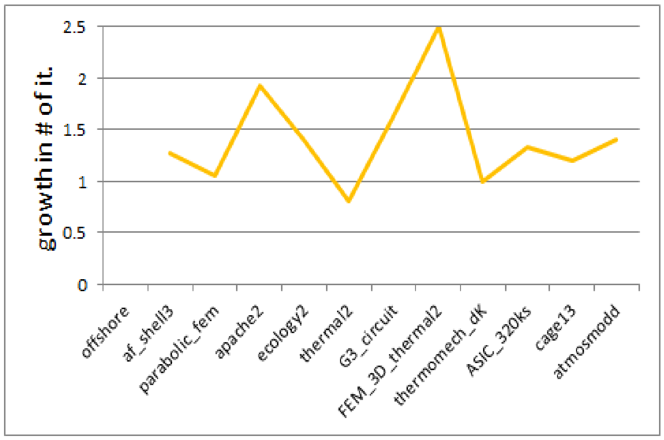


Figure 11: Growth in the number of iterations using graph coloring vs. level-scheduling

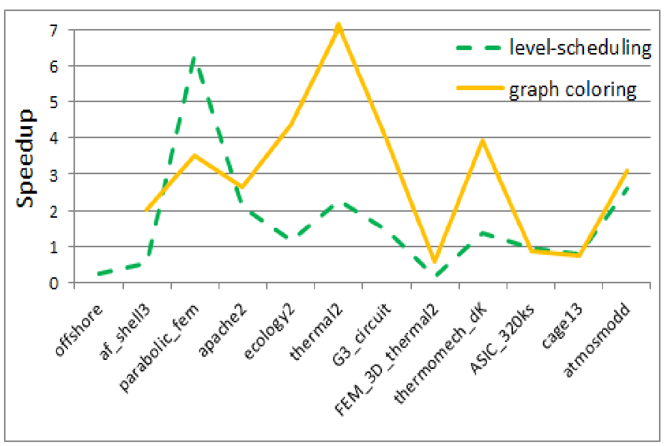


Figure 12: Speedup of iterative method using graph coloring vs. level-scheduling

There are two important takeaways from these experiments. The first is that graph coloring often resulted in an increase in the # of iterations taken to convergence, see Fig. 11. Notice that it is often $< 1.5\times$, but there is a single case indicated by † in Tab. 4 when the iterative method actually failed to converge.

The second takeaway is that in most cases we have obtained a larger speedup using graph coloring when compared to level-scheduling for the overall run time of the iterative method, see Fig. 12. Notice that using graph coloring we have increased the degree of available parallelism, and therefore each iteration is much faster than before. Ultimately, even though we take more iterations, in most cases in our numerical experiments the significant speedup per iteration still allows us to achieve an overall speedup for the iterative method.

This brief empirical study showcases some of the tradeoffs of working with level-scheduling and graph coloring based approaches.