Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU

Maxim Naumov
NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050

Abstract

A novel algorithm for computing the incomplete-LU and Cholesky factorization with 0 fill-in on a graphics processing unit (GPU) is proposed. It implements the incomplete factorization of the given matrix in two phases. First, the symbolic analysis phase builds a dependency graph based on the matrix sparsity pattern and groups the independent rows into levels. Second, the numerical factorization phase obtains the resulting lower and upper sparse triangular factors by iterating sequentially across the constructed levels. The Gaussian elimination of the elements below the main diagonal in the rows corresponding to each single level is performed in parallel. The numerical experiments are also presented and it is shown that the numerical factorization phase can achieve on average more than 2.8× speedup over MKL, while the incomplete-LU and Cholesky preconditioned iterative methods can achieve an average of 2× speedup on GPU over their CPU implementation.

1 Introduction

The solution of large sparse linear systems is an important problem in computational mechanics, geophysics, biology, circuit simulation and many other applications in the field of computational science and engineering. In general, these linear systems can be solved using direct or preconditioned iterative methods. Although the direct methods are often more reliable, they usually have large memory requirements and do not scale well on massively parallel computer platforms.

NVIDIA Technical Report NVR-2012-003, May 2012.

In this paper we focus on the preconditioned iterative methods that are more amenable to parallelism and therefore can be used to solve larger problems. Currently, the most popular iterative schemes belong to the Krylov subspace family of methods. They include Bi-Conjugate Gradient Stabilized (BiCGStab) and Conjugate Gradient (CG) iterative methods for nonsymmetric and symmetric positive definite (s.p.d.) linear systems, respectively. In practice, one often uses a variety of preconditioning techniques to improve the convergence of these iterative methods. In this paper we focus on the incomplete-LU and Cholesky factorization with 0 fill-in, which is one of the most popular of these preconditioning techniques [3, 31].

The parallel implementation of these incomplete factorizations has been studied by many authors, that explore the lack of dependencies between rows in the Gaussian elimination using different reordering techniques. There are three overarching reordering strategies used in these studies.

The first strategy is based on level-set orderings (also called level-scheduling) and is similar to the approach used for the solution of sparse triangular linear systems in [14, 25]. It often consists of an implicit reordering of independent rows into levels and a subsequent sequential traversal of those levels such that the Gaussian elimination within a single level is performed in parallel [13, 23].

The second strategy is based on multi-coloring orderings (also called graph coloring) that results in a permutation that is usually explicitly applied to the matrix at hand. In this approach if k colors are used to color the matrix adjacency graph, then the permuted matrix has k blocks on its diagonal that are themselves diagonal matrices. The Gaussian elimination corresponding to each diagonal block in the permuted matrix can then be performed in parallel [23, 27, 28].

The third strategy is based on the independent set orderings [21] that gives rise to the multi-level incomplete factorization. It consists of an implicit or explicit reordering of the rows resulting in a matrix with a diagonal upper-left block, a step of Gaussian elimination for all the rows corresponding to this block, and then a recursive application of the same algorithm [2, 5, 6, 28, 29, 30]. A similar non-recursive approach can be applied using the hierarchical graph ordering [15].

Finally, we mention that related work on parallelizing the incomplete factorizations has also been done in [1, 16, 17, 19, 24], while the application of some of these techniques to GPUs has been studied in [18, 22].

In these strategies, it is important to keep in mind that implicit reordering techniques have no effect on the convergence of the incomplete-LU and Cholesky preconditioned iterative methods, while the effect of explicit reorderings is non trivial and has been studied in [4, 10] for nonsymmetric and in [9, 11] for s.p.d. problems among many others.

In this paper we follow the first strategy, exploring the available parallelism using an implicit reordering and splitting the factorization into two phases. First, the *symbolic analysis* phase builds a dependency graph based on the matrix sparsity pattern and groups the independent rows into levels. Second, the *numerical factorization* phase obtains the resulting lower and upper sparse triangular factors by iterating sequentially across the constructed levels. The Gaussian elimination of the elements below the main diagonal in the rows corresponding to each single level is performed in parallel.

It is important to notice that in the preconditioned iterative methods the incomplete factorization is computed only once, while the linear systems with the resulting sparse triangular factors are often solved multiple times. Therefore, in order to reduce the overhead of the computationally expensive *symbolic analysis* phase, we must reuse the information obtained in it across both the *numerical factorization* phase of the incomplete factorization and the *solve* phase of the solution of sparse triangular linear systems [25].

Although on the surface the parallel incomplete factorization and solution of sparse triangular linear systems are not related, it turns out that exactly the same algorithm can be used to analyse the parallelism available in both problems. In other words, the *symbolic analysis* phase can be implemented using the modified topological sort, breadth-first-search and other graph search algorithms [7, 8, 12] in the same way as it was done in [25]. The algorithm that constructs the directed acyclic graph that allows us to explore the parallelism in both the *numerical factorization* and *solve* phases will be explained in more details in the next section.

Finally, we mention that the incomplete-LU and Cholesky factorizations with 0 fill-in are implemented using CUDA parallel programming paradigm [20, 26, 32], which allows us to explore the computational resources of the GPU. These new algorithms, the corresponding sparse triangular solve, as well as other standard sparse linear algebra operations are exposed as a set of routines in the CUSPARSE library [34].

Although the parallelism available in these algorithms depends highly on the sparsity pattern of the matrix at hand, in the numerical experiments section it will be shown that the *numerical factorization* phase can achieve on average more than 2.8× speedup over MKL, while the incomplete-LU and Cholesky preconditioned iterative methods can achieve an average of 2× speedup using the CUSPARSE library on the GPU over their MKL implementation on the CPU.

Since the incomplete-LU and Cholesky factorizations with 0 fill-in are very similar, we focus only on the former in the next sections.

2 Symbolic Analysis and Numerical Factorization

We are interested in computing the incomplete-LU factorization with 0 fill-in

$$A \approx LU \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a nonsingular matrix, L and $U \in \mathbb{R}^{n \times n}$ are the resulting lower and upper triangular factors, respectively. In further discussion we denote the elements of the matrix $A = [a_{ij}]$, $L = [l_{ij}]$ and $U = [u_{ij}]$, with $l_{ij} = 0$ for $i < j$ and $u_{ij} = 0$ for $i > j$, respectively. Also, we assume that no pivoting is performed and consequently by the definition of incomplete factorization with 0 fill-in the sparsity pattern of A and $L + U$ is the same.

We can represent the data dependencies in the incomplete-LU factorization of the matrix A as a directed graph, where the nodes represent rows and the arrows represent the data dependencies between them. This directed graph is constructed so that there is an arrow from node j to node i if there is an element $a_{ij} \neq 0$ for $i > j$ present in the matrix. In other words, the dependencies between rows are defined by the sparsity pattern of the lower triangular part and are independent of the upper triangular part of the original matrix A.

Notice that because only the lower triangular part of the matrix is involved in the construction of the graph, there are no circular data dependencies in it, consequently there are no cycles in the graph. Also, notice that because we assume that we are able to successfully obtain the incomplete factorization without pivoting, we implicitly assume that $a_{ii} \neq 0$ for $i = 1, \ldots, n$, in other words, each row contains at least one non-zero element on the matrix main diagonal.

Let us consider the following matrix as an example

$$\begin{pmatrix}
a_{11} & * & * & * & * & * & * & * & * \\
 & a_{22} & * & * & * & * & * & * & * \\
 & & a_{33} & * & * & * & * & * & * \\
a_{41} & & & a_{44} & * & * & * & * & * \\
a_{51} & & & & a_{55} & * & * & * & * \\
 & a_{62} & & & & a_{66} & * & * & * \\
 & & a_{73} & & & & a_{77} & * & * \\
 & & & a_{84} & a_{85} & & & a_{88} & * \\
 & & & a_{94} & a_{95} & & & & a_{99}
\end{pmatrix} \tag{2}$$

where $*$ denotes an element that is either present or not in the matrix.

The directed acyclic graph (DAG) illustrating the data dependencies in the incomplete-LU factorization of the matrix in (2) is shown in Fig. 1. Notice that

it is identical to the DAG that describes the dependencies in the solution of the lower triangular linear system with the coefficient matrix L used in [25]. Also, notice that even though the sparsity pattern in (2) might look sequential at first glance, there is plenty of parallelism to be explored in it.

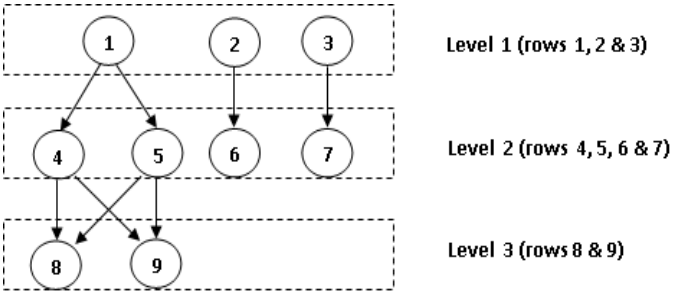

Figure 1: The data dependency DAG of the original matrix A

In practice we do not need to construct the data dependency DAG because it is implicit in the matrix. It can be traversed using for example a modified breadth-first-search (BFS) shown in Alg. 1. 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. 1. This information is passed to the *numerical factorization* phase, which can process the nodes belonging to the same level in parallel.

Algorithm 1 Symbolic Analysis Phase

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

In the *numerical factorization* phase we can explore the parallelism available in each level using multiple threads, but because the levels must be processed sequentially one-by-one, we must synchronize all threads across the level boundaries as shown in Alg. 2.

Algorithm 2 Numerical Factorization Phase

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

Since the sparsity pattern of the original matrix A and $L+U$ is the same and we do not need to store the unitary diagonal of L, the resulting incomplete factorization is stored as a general matrix with its lower and upper triangular parts containing the lower L and upper U triangular factors, respectively. In fact, notice that on line 5 in Alg. 2 we are implicitly assuming that the elements of the original matrix A are being overwritten with the elements of L and U.

Since the *symbolic analysis* phase of the incomplete-LU factorization with 0 fill-in and the *analysis* phase of the sparse lower triangular solve was shown to be the same, and the later has already been described in great detail in [25], in the next section we focus only on the CUDA parallel implementation of the *numerical factorization* phase.

3 Implementation on the GPU

We assume that the matrix and all the intermediate data structures are stored in the device (GPU) memory, with the exception of a small control data structure stored in the host (CPU) memory. Also, we assume that the matrices are stored in the compressed sparse row (CSR) storage format [31].

For example, letting all the elements in (2) marked by $*$ be zero, the matrix would be stored as

$$\begin{aligned} rowPtr &= \begin{pmatrix} 1 & 2 & 3 & 4 & 6 & 8 & 10 & 12 & 15 & 18 \end{pmatrix} \\ colInd &= \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 1 & 5 & 2 & 6 & 3 & 7 & 4 & 5 & 8 & 4 & 5 & 9 \end{pmatrix} \\ Val &= \begin{pmatrix} l_{11} & l_{22} & l_{33} & l_{41} & l_{44} & l_{51} & l_{55} & l_{62} & l_{66} & l_{73} & \dots & l_{99} \end{pmatrix} \end{aligned} \quad (3)$$

The output of the *analysis* phase are the arrays *chainPtrHost*, *levelPtr* and *levelInd* that have the beginning and end of the chains, levels and the list of sorted rows belonging to every level, respectively. The array *chainPtrHost* determines the properties and the number of kernels to be launched in the *numerical factorization* phase and therefore must be present in the host (CPU) memory. It is usually a relatively short array when compared to the matrix size and is the only data structure present in the host memory.

For example, for (3) these arrays are

$$\begin{aligned} chainPtrHost &= \begin{pmatrix} 1 & 4 \end{pmatrix} \\ levelPtr &= \begin{pmatrix} 1 & 4 & 8 & 10 \end{pmatrix} \\ levelInd &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \end{aligned}$$

Notice that in this particular example there are only a few rows belonging to every level, so that all of the levels are linked into a single chain, and consequently can be processed with a single kernel in the *numerical factorization* phase.

The *numerical factorization* phase accepts as an input a set of levels, the sorted list of rows belonging to every level and the *chain* data structure. It determines the optimal number of thread blocks b needed to process each chain and launches a single $b = 1$ or multiple $b > 1$ thread block kernels in a loop until all chains have been processed. Notice that multiple thread blocks kernel always processes a single level, while a single thread block kernel can process one or more levels.

It is worth mentioning that in general rows might be grouped into levels without preserving their original ordering, in other words, an earlier row can be assigned to a latter level (consider for example the coefficient matrix in (2) augmented with an extra row with a single diagonal element in it). Although we do not have control over assignment of rows across levels, the sorting of rows within a level that is done in the *analysis* phase improves the coalescing of memory reads without affecting parallelism.

The resulting *numerical factorization* pseudo-code is shown in Alg. 3-4.

Algorithm 3 Numerical Factorization Phase

1: Let b and k be the number of thread blocks and chains, respectively.
2: Let $levelInd[]$ contain the list of rows in each level.
3: Let $levelPtr[]$ contain the starting index (into the array $levelInd$) of each level (and an extra element to indicate the end of the last level).
4: Let $chainPtrHost[]$ contain the starting index (into the array $levelPtr$) of each chain (and an extra element to indicate the end of the last chain).

Algorithm 4 Numerical Factorization Phase (Part 2)

```
5: for i ← 0, k do                                          ▷ Process the Chains
6:     start ← chainPtrHost[i]
7:     end   ← chainPtrHost[i + 1]
8:     if single block is enough then
9:         PROCESS_LEVEL_SINGLEBLOCK<<< 1, ... >>>(start, end)
10:    else
11:        PROCESS_LEVEL_MULTIBLOCK<<< b, ... >>>(start)
12:    end if
13: end for

14: procedure PROCESS_LEVEL_SINGLEBLOCK(start, end)          ▷ CUDA Kernel
15:    for e ← start, end do
16:        ...
17:        __syncthreads()
18:    end for
19: end procedure

20: procedure PROCESS_LEVEL_MULTIBLOCK(e)                    ▷ CUDA Kernel
21:    ...
22: end procedure
```

Now let us focus our attention on the update of the elements within a single row. Here we are concerned only with the rows that have at least one element to the left of the main diagonal, otherwise nothing needs to be done for this row.

We will call the row where the elements are being updated the *current* row and the row which is being scaled and added to it, in order to create zeros to the left of the main diagonal, the *reference* row. Since there is always a diagonal element in the *reference* row, there is always at least one element that needs to be updated in the *current* row, while the other elements are updated only if there is an intersection between the sparsity pattern of the remainder of the *current* and *reference* rows.

Notice that based on the number of elements to the left of the main diagonal, we might need to update the same elements multiple times in the *current* row. If multiple updates to the *current* row are needed, the sparsity pattern of the upper triangular part of the original matrix A dictates whether these updates can be performed in parallel.

Letting (a_{ij}, a_{ik}) with $j < k < i$ be a pair of two elements in the i-th row in the lower triangular part of the matrix, there are three cases that can be considered for the corresponding row updates

- The updates to the elements of the i-th row must be performed sequentially if element a_{jk} is present in the upper triangular part of the matrix.
- Otherwise, the updates may be performed in parallel
  i. using atomic floating point operations to update the overlapping elements if the remaining sparsity pattern of rows i, j and k does overlap.
  ii. completely independently with respect to each other, if the remaining sparsity pattern of rows i, j and k does not overlap.

However, we must perform additional analysis of the original matrix in order to take advantage of the parallelism at this stage of the algorithm, which may not be worthwhile. In this paper for simplicity we will always assume that multiple updates to the *current* row are performed sequentially.

Finally, the pseudo-code for updating the elements corresponding to a single row with at least one element to the left of the diagonal is given in Alg. 5. Notice that the loop corresponding to multiple updates to the *current* row on line 7 in Alg. 5 is indeed performed sequentially.

Algorithm 5 Update Elements in the Row

```
1: Let row be the current row.
2: Let Val[] contain the list of values in each row.
3: Let colInd[] contain the list of sorted column indices in each row.
4: Let rowPtr[] contain the starting index (into the array colInd) of each row
   (and an extra element to indicate the end of the last row).
5: start ← rowPtr[row]
6: end   ← rowPtr[row + 1]
7: for j ← start, end do                ▷ Left to Right, until Diagonal Is Reached
8:     col ← colInd[j]
9:     if col < row then
10:        ref ← col                                  ▷ Save the Reference Row
11:        a_{row,col} ← Val[j]
12:        a_{row,ref} ← a_{row,col} / a_{ref,ref}    ▷ Compute and Store the Multiplier
13:        for k ← j + 1, end in parallel do          ▷ Update the Leftover
14:            col ← colInd[k]
15:            if a_{ref,col} ≠ 0 then    ▷ Exists the Element in the Reference Row
16:                a_{row,col} ← Val[k]   ▷ Update the Element in the Current Row
17:                a_{row,col} ← a_{row,col} − a_{row,ref} × a_{ref,col}
18:            end if
19:        end for
20:    end if
21: end for
```

4 Numerical Experiments

In this section we study the performance of the incomplete-LU and Cholesky factorization with 0 fill-in as a standalone algorithm and as a part of a preconditioned iterative method. We use twelve matrices selected from The University of Florida Sparse Matrix Collection [36] 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. 1.

#	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 1: Symmetric positive definite (s.p.d.) and nonsymmetric test matrices

In the following experiments we use the hardware system with NVIDIA C2050 (ECC on) GPU and Intel Core i7 CPU 950 @ 3.07GHz, using the 64-bit Linux operating system Ubuntu 10.04 LTS, CUSPARSE library 5.0 and MKL 10.2.3. The environment variables MKL_NUM_THREADS and KMP_AFFINITY are set to 4 and "granularity=fine,proclist=[0,1,2,3],explicit", allowing MKL to use 4 threads pinned to the corresponding 4 CPU cores.

4.1 Incomplete-LU and Cholesky Factorization (Standalone)

Let us first analyse the performance of the standalone incomplete-LU and Cholesky factorization with 0 fill-in on the GPU, which will be denoted `ilu0` and `ic0`, respectively. We will compare it to the performance attained by the MKL `csrilu0` routine, which will also be used for s.p.d. matrices on the CPU (because of a lack of a corresponding `csric0` routine in MKL).

The absolute time in seconds (s) taken to perform the incomplete-LU and Cholesky factorization on the CPU using the MKL `csrilu0` routine and on the GPU using the CUSPARSE library `csrsv_analysis` and `ilu0/ic0` routines is given in Tab. 2. The total time taken by the CUSPARSE library incomplete factorization is the sum of the time taken by the *symbolic analysis* and the *numerical factorization* phases performed by `csrsv_analysis` and `ilu0/ic0` routines, respectively. The fact that on the CPU we use the MKL `csrilu0` routine for the s.p.d. matrices as well as the nonsymmetric matrices is denoted by †. Contrary to the usual expectations this setup is less advantageous for the CUSPARSE library, because on the GPU the incomplete-Cholesky factorization is roughly 30% slower than the incomplete-LU factorization for the same matrix.

	CUSPARSE			MKL
#	`csrsv_analysis` time (s)	`ilu0/ic0` time (s)	total time (s)	`csrilu0` time (s)
1.	0.0594	0.3030	0.3695	0.3448†
2.	0.1006	0.6048	0.6148	1.6049†
3.	0.0362	0.0187	0.0549	0.1123†
4.	0.0480	0.0470	0.0950	0.0842†
5.	0.0669	0.0496	0.1165	0.0691†
6.	0.0837	0.1023	0.1860	0.3521†
7.	0.1037	0.0865	0.1902	0.1537†
8.	0.0599	0.3225	0.3824	0.2831
9.	0.0342	0.0622	0.0964	0.1770
10.	0.0270	0.1730	0.2000	0.0982
11.	0.0656	0.0847	0.1503	0.6826
12.	0.1051	0.0430	0.1481	0.1781

Table 2: Time taken by MKL `csrilu0` and CUSPARSE `csrsv_analysis` and `ilu0/ic0`

The total time taken by the CUSPARSE library and MKL to compute the incomplete factorization is summarized in Fig. 2. Notice that there is a slight variation in the time taken by the `csrsv_analysis` routine with respect to the previous results in [25]. It is due to the new version of the CUSPARSE library where the sparse triangular solve now operates on the lower and upper triangular parts of the full matrix (instead of the individually stored lower and upper triangular factors).

Although MKL often outperforms the CUSPARSE library if we consider the total time taken by the incomplete factorization, recall that the *symbolic analysis* phase is shared between the *numerical factorization* phase and the

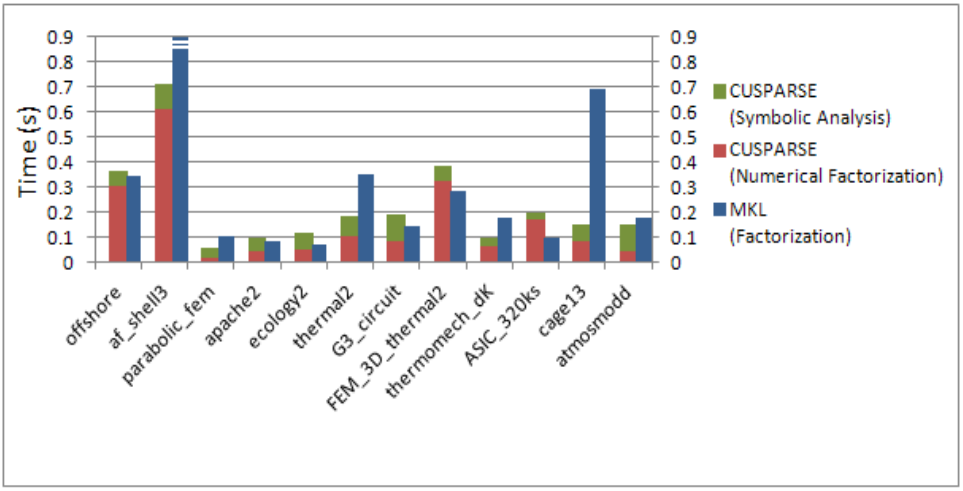

Figure 2: CUSPARSE (*symbolic analysis* & *numerical factorization*) and MKL (fact.) time

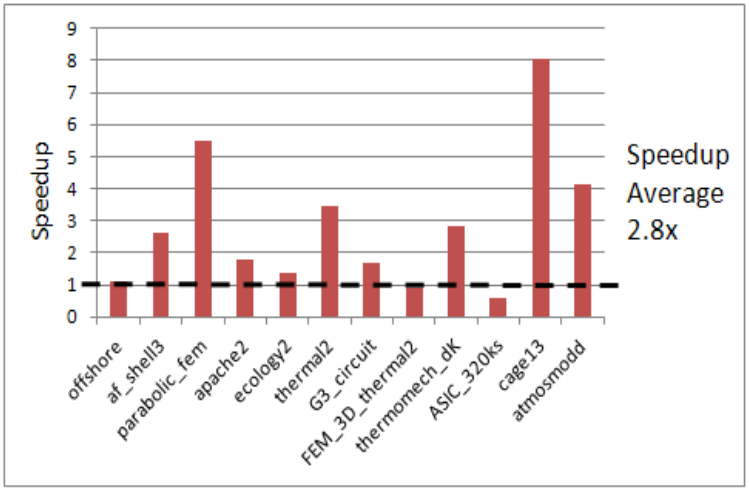

Figure 3: Speedup of CUSPARSE `ilu0/ic0` versus MKL `csrilu0`

solve phase of the solution of sparse triangular linear systems. It has already been shown that when the solution of sparse triangular linear system needs to be performed multiple times, the cost of the slower *symbolic analysis* phase can be amortized across multiple faster applications of the *solve* phase, resulting in an average 2× speedup [25]. In this setting we must already perform the *symbolic analysis* for the sparse triangular solve, therefore from the incomplete

factorization standpoint it is free and we can focus only on the time taken by the *numerical factorization* phase. For this reason, we highlight in red the fastest time between CUSPARSE `ilu0/ic0` and MKL `csrilu0` in Tab. 2 and show the corresponding speedup in Fig. 3.

4.2 Incomplete-LU and Cholesky Factorization (Iterative Method)

Let us now analyse the performance of the incomplete factorization, in the context of solving the linear system

$$A\mathbf{x} = \mathbf{f} \tag{4}$$

using preconditioned Bi-Conjugate Gradient Stabilized (BiCGStab) and Conjugate Gradient (CG) iterative methods for nonsymmetric and s.p.d. systems, respectively. We precondition these methods using the incomplete-LU $A \approx LU$ in (1) and Cholesky $A \approx R^T R$ factorizations with 0 fill-in.

We compare the implementation of the BiCGStab and CG iterative methods 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.

	CPU				GPU				Speedup
#	ilu0 time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	ilu0/ic0 time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	
1	0.35	0.68	8.83E-08	25	0.30	1.57	8.83E-08	25	0.55
2	1.59	38.1	9.88E-08	571	0.60	35.1	9.88E-08	570	1.11
3	0.11	31.1	9.84E-08	1044	0.02	7.48	9.84E-08	1044	4.16
4	0.01	31.9	9.97E-08	713	0.05	13.4	9.97E-08	713	2.37
5	0.07	105.	9.98E-08	1746	0.05	57.0	9.98E-08	1746	1.84
6	0.36	142.	9.99E-08	1655	0.10	56.8	9.92E-08	1655	2.50
7	0.16	18.3	8.86E-08	183	0.09	9.02	8.22E-08	183	2.03
8	0.28	0.16	5.25E-08	4	0.32	0.53	5.25E-08	4	0.52
9	0.18	88.1	1.57E-04	2000	0.06	50.0	1.97E-04	2000	1.76
10	0.1	0.24	6.33E-08	6	0.17	0.15	6.33E-08	6	1.06
11	0.67	0.25	2.52E-08	2.5	0.08	0.23	2.52E-08	2.5	2.97
12	0.19	11.8	9.58E-08	75.0	0.04	4.81	7.92E-08	73.5	2.47

Table 3: The `ic0` preconditioned CG and `ilu0` preconditioned BiCGStab methods

The results of the numerical experiments are shown in Tab. 3, where we state the speedup obtained by the iterative method on the GPU over CPU (speedup), 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 factorization (`ilu0/ic0`) and iterative solution of the linear system (solve). We include the time taken by factorization on the GPU and CPU, but we exclude the extra time taken to transform the incomplete-LU upper triangular factor U into the incomplete-Cholesky upper triangular factor R for s.p.d. matrices on the CPU, in the computed speedup.

Finally, the speedup based on the total time taken by the preconditioned iterative method is summarized in Fig. 4, where * indicates that the method did not converge to the required tolerance.

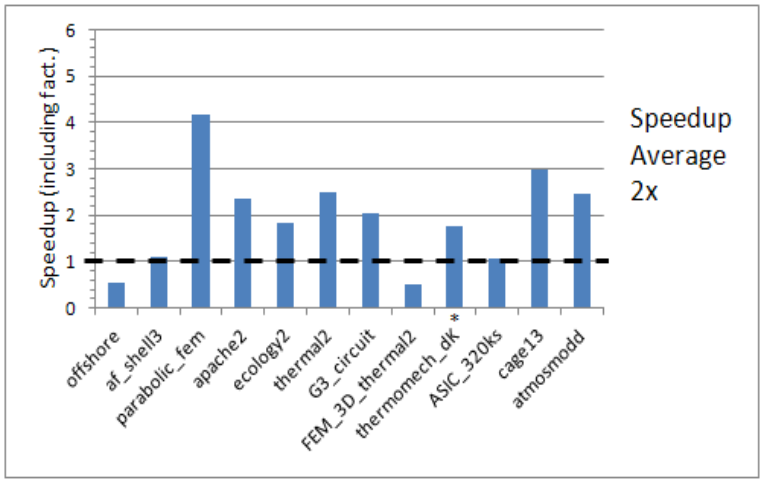

Figure 4: Speedup of BiCGStab and CG with incomplete-LU/Cholesky preconditioning

Notice that for most of the matrices the total speedup has not changed significantly when compared to the earlier results [25], where the incomplete factorization was performed on the CPU. This is not surprising given that in our numerical experiments the incomplete factorization often consumes only a small fraction of the total time taken by the iterative method. However, as shown on Fig. 5 there are matrices for which this is not the case. In particular, performing the incomplete-LU factorization on the GPU for cage13 allowed us to significantly reduce the overall time taken by the iterative method.

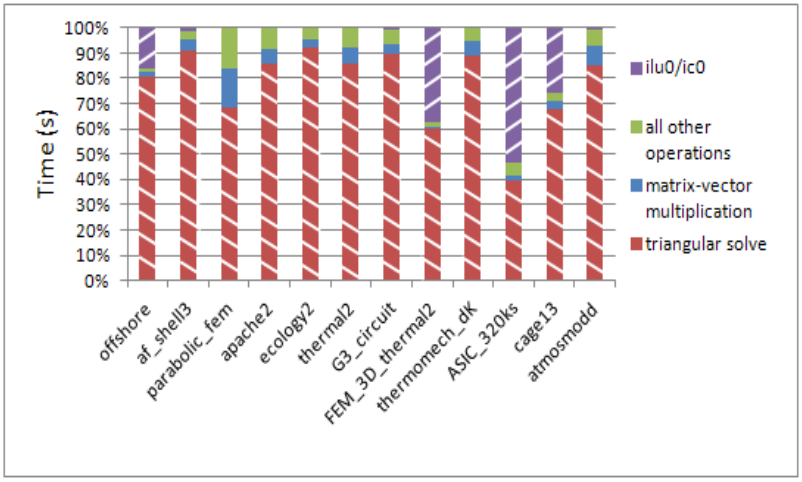

Figure 5: Profiling of BiCGStab and CG with incomplete-LU/Cholesky preconditioning

Notice that for majority of matrices in our numerical experiments the implementation of the iterative method using the CUSPARSE and CUBLAS libraries does indeed outperform the MKL. In fact the *numerical factorization* phase of the incomplete factorization obtains on average more than 2.8× speedup over MKL and the incomplete-LU and Cholesky preconditioned BiCGStab and CG iterative methods obtain an average of 2× speedup on the GPU over their CPU implementation.

5 Conclusion

A novel parallel algorithm for computing the incomplete-LU and Cholesky factorization with 0 fill-in was developed. It splits the incomplete factorization in two phases. The *symbolic analysis* phase, that is the same as the *analysis* phase of the solution of sparse triangular linear systems, and the *numerical factorization* phase. The performance of the incomplete factorization depends highly on the sparsity pattern of the matrix at hand. In general, the sparsity of the lower triangular part defines which rows can be processed independently, while the sparsity of the upper triangular part defines which updates to a single row can be made in parallel. Although, there are sparsity patterns for which the computation is inherently sequential, there are many other realistic sparsity patterns where enough parallelism is available.

The new algorithm is ideally suited for the incomplete-LU and Cholesky preconditioned iterative methods. In this setting the CUDA implementation of the *numerical factorization* phase on the GPU can outperform the MKL implementation on the CPU, while the computational cost of the *symbolic analysis* phase can be shared with *analysis* phase of the solution of sparse triangular linear systems and amortized across multiple steps of an iterative method.

In our numerical experiments the *numerical factorization* achieved on average more than 2.8× speedup over MKL, while the incomplete-LU and Cholesky preconditioned iterative methods implemented on the GPU using the CUSPARSE and CUBLAS libraries achieved an average of 2× speedup over their MKL implementation. To conclude, it is worth mentioning that the use of multiple-right-hand-sides would increase the available parallelism and can result in a significant relative performance improvement on the GPU.

6 Acknowledgements

The author would like to acknowledge Ujval Kapasi and Philippe Vandermersch for their useful comments and suggestions.

References

[1] J. I. Aliaga, M. Bollhofer, A. F. Martin, E. S. Quintana-Orti, "Exploiting Thread-level Parallelism in the Iterative Solution of Sparse Linear Systems", Parallel Comput., Vol. 37, pp. 183-202, 2011.

[2] R. Bank and C. Wagner, "Multilevel ILU decomposition", Numer. Math., Vol. 82, pp. 543-576, 1999.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, PA, 1994.

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

[5] 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.), Vol. 37, pp. 1294-1300, 1997.

[6] 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.

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

[8] T. A. Davis, "Direct Methods for Sparse Linear Systems", SIAM, Philadelphia, PA, 2006.

[9] I. S. Duff and G. A. Meurant, "The Effect of Ordering on Preconditioned Conjugate Gradients", BIT, Vol. 29, pp. 635-657, 1999.

[10] L. C. Dutto, "The Effect of Ordering on Preconditioned GMRES Algorithm for Solving the Compressible Navier-Stokes Equations", Internat. J. Numer. Methods Eng., Vol. 36, pp457-497, 1993.

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

[12] R. K. Ghosh and G. P. Bhattacharjee, "A Parallel Search Algorithm for Directed Acyclic Graphs", BIT, pp. 134-150 (24), 1984.

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

[14] P. Gonzalez, J. C. Cabaleiro and T. F. Pena, "Solving Sparse Triangular Systems on Distributed-Memory Multicomputers", IEEE Euromicro (PDP98), 1999.

[15] P. Henon and Y. Saad, "A Parallel Multistage ILU Factorization Based on a Hierarchical Graph Decomposition", SIAM J. Sci. Comput., Vol. 28, pp. 2266-2293, 2006.

[16] D. Hysom and A. Pothen, "A Scalable Parallel Algorithm for Incomplete Factor Preconditioning", SIAM J. Sci. Comput., Vol. 22, pp. 2194-2215, 2001.

[17] D. Hysom and A. Pothen, "Level-based Incomplete LU factorization: Graph Model and Algorithms", Technical Report UCRL-JC-150789, Lawrence Livermore National Labs, 2002.

[18] V. Heuveline, D. Lukarski and J.-P. Weiss, "Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method", EMCL Preprint Series, 2191-0693, KIT, 2011.

[19] G. Karypis and V. Kumar, "Parallel Threshold-based ILU Factorization", Proc. ACM/IEEE Supercomputing (SC97), pp. 28, 1997.

[20] D. B. Kirk and W. W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach", Elsevier, 2010.

[21] R. Leuze, "Independent Set Orderings for Parallel Matrix Factorizations by Gaussian Elimination", Parallel Comput., Vol. 10, pp. 177-191, 1989.

[22] R. Li and Y. Saad, "GPU-accelerated Preconditioned Iterative Linear Solvers", Technical Report UMSI-2010-112, Minnesota Supercomputer Institute, 2010.

[23] S. Ma and Y. Saad, "Distributed ILU(0) and SOR Preconditioners for Unstructured Sparse Linear Systems", Technical Report UMSI-94-101, University of Minnesota Supercomputer Institute, 1994.

[24] M. Magolu Monga Made and H. A. van der Vorst, "Parallel Incomplete Factorizations with Pseudo-overlapped Subdomains", Parallel Comput., Vol. 27, pp. 989-1008, 2001.

[25] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU", Technical Report NVR-2011-001, NVIDIA, 2011.

[26] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA", Queue, pp. 40-53 (6-2), 2008.

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

[28] Y. Saad, "ILUM: A Multi-Elimination ILU Preconditioner for General Sparse Matrices", SIAM J. Sci. Comput., Vol. 17, pp. 830-847, 1996.

[29] Z. Li, Y. Saad and M. Sosonkina, "pARMS: A Parallel Version of the Algebraic Recursive Multilevel Solver", Numer. Linear Algebra Appl., Vol. 10, pp. 485-509, 2003.

[30] Y. Saad and B. Suchomel, "ARMS: An Algebraic Recursive Multilevel Solver for General Sparse Linear Systems", Numer. Linear Algebra Appl., Vol. 9, pp. 359-378, 2002.

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

[32] J. Sanders and E. Kandrot, "CUDA by Example: An Introduction to General-Purpose GPU Programming", Addison-Wesley, 2010.

[33] C. F. van Loan and G. H. Golub, "Matrix Computations", The John Hopkins University Press, 3rd Ed., 1996.

[34] "NVIDIA CUSPARSE and CUBLAS Libraries", http://www.nvidia.com/object/cuda_develop.html

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

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