A synchronization-free incomplete LU factorization for GPUs with level-set analysis

Manuel Freire
Instituto de Computación
Facultad de Ingeniería - UDELAR
Montevideo, Uruguay
mfreire@fing.edu.uy

Ernesto Dufrechou
Instituto de Computación
Facultad de Ingeniería - UDELAR
Montevideo, Uruguay
edufrechou@fing.edu.uy

Pablo Ezzatti
Instituto de Computación
Facultad de Ingeniería - UDELAR
Montevideo, Uruguay
pezzatti@fing.edu.uy

***Abstract*—Incomplete factorization methods are powerful algebraic preconditioners widely used to accelerate the convergence of linear solvers. The parallelization of ILU methods has been extensively studied, particularly for GPUs, which are ubiquitous parallel computing devices. In recent years, synchronization-free methods have become the mainstream approach for solving sparse triangular linear systems.**

Although the sparse triangular solver and ILU factorization are closely related, the application of synchronization-free strategies to ILU factorization has not been explored in the literature to the same extent as the triangular solver. In this work, we present synchronization-free implementations of the ILU-0 preconditioner on GPUs. Specifically, we propose three implementations that vary in how row updates are handled after each coefficient elimination, as well as an additional approach that leverages a prior level-set analysis to optimize the execution schedule.

***Index Terms*—ILU, CSR, Synchronization Free**

I. Introduction

Solving large-scale linear systems of equations, $Ax = b$, where A is a sparse matrix and b is a known vector, is a fundamental problem in scientific computing, engineering simulations, optimization, and machine learning. Developing efficient methods to tackle this problem on modern parallel computing platforms is therefore essential.

When solving linear systems directly via methods such as Gaussian elimination or LU decomposition, computational costs and memory requirements grow rapidly with the matrix size, particularly in the case of sparse systems. For large matrices, direct solvers become impractical, and iterative methods like the Conjugate Gradient (CG) method or the Generalized Minimal Residual (GMRES) method [1] are commonly used. However, the convergence rate of many iterative methods can be slow when the matrix A is ill-conditioned (i.e., when the ratio of the largest to the smallest eigenvalue is large), making preconditioning necessary. A preconditioner M is a matrix (or operator) that approximates the matrix A or its inverse in a way that reduces the condition number of the preconditioned system $M^{-1}Ax = M^{-1}b$ significantly compared to the original system.

One such preconditioner is Incomplete LU factorization (ILU) [2], which approximates the matrix A by the product of a lower triangular matrix L and an upper triangular matrix U, such that $M = LU$ and $LU \approx A$. Unlike complete LU decomposition, which computes the full factorization of A, ILU only performs an incomplete factorization by discarding certain fill-ins that would otherwise appear in L and U. This approach preserves sparsity in the factors, helping to control memory usage and computational costs.

ILU is a widely used algebraic preconditioner and is often chosen when no further information about the problem is available. However, ILU factorizations can be computationally expensive, especially for large sparse matrices, partly because ILU parallelism is limited by serial dependencies in the Gaussian elimination sequence. To address this, various efforts have been made to parallelize the ILU on GPUs, including approaches based on level-set analysis [3], [4], graph-coloring [5], and iterative methods [6], [7].

Synchronization-free methods for solving sparse triangular linear systems [8] (the $sptrsv$ operation) have gained popularity in recent years, becoming the standard for implementing these operations on GPUs [9]–[12]. While the analysis of major vendor libraries (such as `cusparse`) suggests that synchronization-free strategies are used for both $sptrsv$ and ILU, scientific literature exploring this paradigm's application to ILU remains limited.

In this work, we propose a row-based parallel implementation of the ILU(0) preconditioner (resulting from discarding all fill-in elements during factorization) based on the synchronization-free strategy for GPUs. We present two variants: one that operates without a prior analysis stage and another that leverages level-set information to improve execution scheduling during computation. Additionally, we provide a detailed algorithm description and explore various implementation options for the most computationally intensive stage.

The rest of the article is organized as follows: Section II reviews key concepts related to ILU factorization and its parallel implementation on GPUs. Our proposal is then detailed in Section IV. Section V presents the experimental evaluation of the new routines, and Section VI concludes the article with remarks and potential directions for future research.

II. Background concepts

The Incomplete LU (or incomplete Cholesky for symmetric positive definite matrices) approximates a matrix A by the

product of a unit lower triangular matrix L and an upper triangular matrix U with a sparsity pattern similar to that of the lower and upper parts of A. It is widely used as an algebraic preconditioner in combination with iterative methods to solve sparse linear systems.

The procedure is essentially an LU factorization without pivoting, where the fill-in elements that appear during the elimination process are discarded based on certain criteria. Allowing more fill-in generally improves the accuracy of the approximation $LU \approx A$. However, an effective preconditioner must remain lightweight, so that the product with its inverse (or in this case, solving the two triangular systems) can be efficiently computed in each iteration of the linear solver. Therefore, different strategies for allowing fill-in elements in the factors are employed, leading to various ILU variants.

The most straightforward of these variants is the one that discards all fill-in elements, i.e., the sparsity pattern $\mathcal{S}(L + U) = \mathcal{S}(A)$, which is known as ILU(0) or ILU-0.

Algorithm 1 describes the ILU-0 for a sparse matrix A, which is replaced by the factors L and U at the end of the computation. As in the full LU decomposition, there are several ways to traverse the matrix during computation, yielding the right-looking, left-looking, or Crout's variant. In this case, we focus on a row-based variant, as it is well-suited for the CSR sparse matrix format.

Algorithm 1 In-place row-based variant of the ILU-0

Require: A
Ensure: L as the strict lower triangle of A and U as the upper triangle of A
```
for i = 2:n do
  for k = 1:i-1 do
    if a_ik ≠ 0 then
      a_ik = a_ik / a_kk
    end if
    for j = k+1:n do
      if a_ij ≠ 0 then
        a_ij = a_ij − a_ik a_kj
      end if
    end for
  end for
end for
```

It can be noted from Algorithm 1 that dependencies exist between the computations. Specifically, a_{ik} and a_{kk} must be finalized before a_{ik} is replaced with a_{ik}/a_{kk}. Although this condition is more restrictive than necessary, for simplicity, this is typically enforced by computing $a_{ik} = a_{ik}/a_{kk}$ only after row k is fully processed. This still allows multiple rows to be computed in parallel.

Furthermore, if a_{ij} with $j < k$ is nonzero and a_{jk} is also nonzero, then a_{ij} and a_{ik} cannot be computed in parallel. This restricts the parallel processing of elements within the same row. This parallelism is less practical to exploit because it requires analyzing each row to identify groups of coefficients that can be processed concurrently. As a result, the elements in the lower part of each row are typically processed sequentially.

The dependencies described above can be represented using a Directed Acyclic Graph (DAG). The adjacency matrix of this graph is determined by the lower triangle of A and is identical to that of the corresponding triangular linear system. This connection between the solution of triangular systems and the ILU factorization suggests that techniques for handling dependencies in `sptrsv` could also be effective for ILU.

The two main strategies for parallelizing `sptrsv` on `GPU` are level-set-based methods and self-scheduled (or synchronization-free) methods. In the self-scheduled approach, parallel operations are determined dynamically during the solution process, while the level-set-based method uses a preprocessing phase to establish an execution schedule. As a result, level-set-based methods divide `sptrsv` into two phases: an analysis phase, performed once per nonzero pattern, and a solving phase, which varies for each right-hand side.

A. Level-set scheduling

The level-set approach involves organizing the equations (or rows of the sparse matrix) into a series of sets. The equations in the first set are independent of others and can be solved immediately. Once the first set has been processed, all equations in the second set can be solved concurrently, and so on.

The sets are formed during the analysis phase, which identifies the dependencies of each row. In the solving phase, any row that is independent of others at a given stage can be processed, i.e., its unknowns can be substituted with the corresponding values. After these equations are solved, the procedure proceeds with the updated set of dependencies.

Traditional implementations of the level-set scheduling employ a barrier to prevent the processing of equations in the next level until all equations in the current level have been resolved. A drawback of this approach is that rows dependent on only a few equations from the previous level may be delayed by the completion of the entire level. For example, in Figure 2, row 4 must wait for rows 2 and 5 to finish, even though it does not directly depend on them, because they belong to lower levels.

$$\begin{bmatrix} 1 & & x & & x & x & \\ & 2 & & x & x & & \\ 1 & & 3 & x & & x & \\ 1 & & 3 & 4 & x & & \\ & 2 & & & 5 & x & x \\ & 2 & 3 & & & 6 & \\ 1 & & & & 5 & 6 & 7 \end{bmatrix}$$

Fig. 1. Example sparse matrix. The values not shown are zeros.

B. Synchronization-free scheduling

An alternative and widely used method for solving sparse triangular linear systems is the synchronization-free (or self-scheduled) algorithm. This approach addresses the synchronization issue by treating the rows as a collection of tasks,

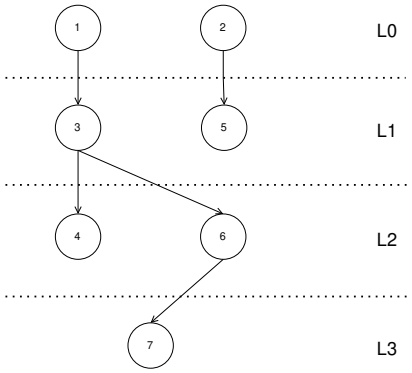

Fig. 2. Level structure generated from the example matrix using the level-set strategy.

each with its own set of dependencies. When a thread becomes available, it selects a task from the shared pool and checks for its dependencies. Additional data structures are required to track the rows that have been processed and the values of the solved unknowns.

In contrast to the level-set-based approach, the synchronization-free paradigm avoids imposing a fixed execution order, thus eliminating the need for matrix preprocessing. Moreover, it processes rows independently of their assigned level. For example, in the matrix shown in Figure 1, row 4 only depends on row 3 (which in turn depends on row 1) and does not wait for rows 2 or 5. An illustration of this execution flow can be found in Figure 3.

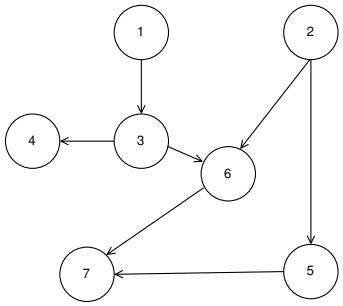

Fig. 3. Dependencies using the sync-free strategy for the matrix of Figure 1. Redundant dependencies were removed

III. Related work

One of the first successful parallel implementations of the Incomplete LU factorization was developed by Hysom and Pothen [3]. Their parallelization is based on the level-set strategy and was included in the *hypre* library. A decade later, in the early years of GPUs as general-purpose massively parallel devices, Naumov presented a level-set-based Incomplete Cholesky factorization for GPUs in [4]. In this work, the level-set structure of the matrix is obtained through the algorithm and data structures described in [13], with synchronization between levels achieved by assigning each level to an individual GPU kernel. Naumov reported average speedups of nearly $3\times$ relative to the Intel MKL library.

Later, in [5], Naumov described a GPU implementation of ILU-0 using graph coloring to extract parallelism and compared it with the level-set strategy. Although the results favored the graph-coloring scheme, it is important to note that this strategy requires an explicit reordering of the matrix, which can affect the convergence of the preconditioned iterative method.

In 2015, Chow and Patel proposed an entirely different approach to computing the ILU factorization [6], based on a fixed-point iteration that allows each element of the factors to be computed in parallel and asynchronously. Chow, Anzt, and Dongarra later presented a GPU implementation of this approach in [7].

In 2016, Liu et al. proposed an `sptrsv` algorithm for GPUs that reduced much of the synchronization cost in Naumov's implementation [9], [14].

In [15], a synchronization-free algorithm for `CSR` was proposed, which addressed some performance challenges of [9], such as the extensive use of atomic operations. The same authors later proposed a synchronization-free strategy for generating the level structure for level-set solvers [12], [16]. In [11], they combined the level-set and synchronization-free approaches into a two-step routine, where the level-set analysis is used to improve row assignment to GPU warps.

Although synchronization-free methods have become the mainstream approach for solving sparse triangular linear systems, and it is likely that the `cusparse` library uses this approach for both `sptrsv` and ILU [17], there are not many recent publications that discuss a synchronization-free ILU implementation. One of the most closely related works is [18], which describes a synchronization-free left-looking LU factorization.

IV. Proposal

The algorithms that we propose in this work use a synchronization-free approach, which is the state-of-the-art in the resolution of triangular systems, to compute the ILU factorization of a sparse matrix in `CSR` format. The general idea is to schedule $warps$ to compute the factorization of each row as soon as the dependencies of that row are processed. Therefore, each $warp$ processes one row of the matrix (which we call the *current* row of the warp following Naumov's naming [4]), iterating on the non-zeros of the lower part of the row, and busy waiting until the row that corresponds to each of them (which we call *reference* row) is processed by another warp. Once each reference row is processed the lower nonzero is divided by the diagonal entry of the reference row and the upper (right-hand) part of the current row gets updated where the nonzero patterns of the current and reference row overlap.

The busy-waiting procedure consist in iteratively polling the corresponding entries of the vector *piv_idx* which, when it has a valid value, stores the index of the diagonal entry of the corresponding row on the `CSR` structure. This value is important after the busy-waiting stage since the accumulation is only done for the values at the right-hand side of the pivot. Besides, it allows to access the diagonal entry directly when

dividing to compute the L factor. We note that finding the pivot in a row is $O(n)$ with n being the number of non-zeros of the row.

The main difference with the $sptrsv$ is the update stage, which is also the main bottleneck of the algorithm. For each lower nonzero, this stage compares the nonzero patterns of the current and reference rows by checking the corresponding column indices of each element.

We propose four different algorithms. All the different variants have the main structure, which we summarize in Algorithm 4. Since the main bottleneck is the update phase, the threads in the $warp$ iterate together through the elements in the lower part of the current row, waiting for the same dependencies and compute the update in parallel after the dependence is met. The first three variants differ in how this parallel row update is performed.

Algorithm 2 General structure of the ILU variants

```
1  // IN/OUT: row_ptr[], col_idx[], val[], is_solved
2
3  init()
4  row = getRowNumber(warp)
5  piv = calculatePiv(row,row_ptr[])
6  li = row_ptr[row]
7  while(li < piv)
8     ref_row=col_idx[li];
9     ready = is_solved[ref_row]
10    if(ready != 0) //ready is piv[ref_row]+1
11       if(warp.rank()==0) val[li]/=val[ready-1];
12       warp.sync()
13    endif
14    update_row(curr_row, ref_row, li, ready-1,
          row_ptr[], col_idx[], val[])
15    li++
16    ready = 0
17 endwhile
18 if(warp.rank()==0)
19    is_solved[curr_row] = piv+1
20 end
```

A. Update by intersecting vectors in shared memory (ILU_{shared})

The first variant, which we call ILU_{shared}, executes the update phase in shared memory, computing the intersection between the column numbers of the nonzeros in the current and reference rows by adapting the ideas in [19]. The threads of the $warp$ work together by iteratively loading the reference row column indices (and values) into shared memory in batches of 32 elements, comparing with the 32 indices of the current row stored in the warp's registers and accumulating in the corresponding elements. After all elements that need it are updated, the head of the vectors is advanced depending on the numerical values of the last entry. The simplified source code of this stage is provided in 3.

B. Update with binary search (ILU_{vect})

Since ILU_{shared} compares, in the worst case, each nonzero of the current row with every element of the reference row, it has a complexity of O($nnz_{curr} \times nnz_{ref}$) which in large rows

Algorithm 3 Intersecting two increasingly sorted vectors using shared memory. Vector a is temporarily stored in registers, while vector b is in shared memory.

```
1  int i_a, int i_b // start of the vectors to
       intersect
2  int l_a, int l_b // length of the vectors
3  VALUE_TYPE piv // current pivot element
4  int * col_idx, volatile VALUE_TYPE * val // CSR
       sparse matrix
5  int lne // thread lane
6  int * idx_sh // shared memory buffer for indices
7
8  while(i_a < l_a && i_b < l_b)
9
10  // 1. Load the vectors (32)
11  my_idx  =(i_a+lne < l_a)?col_idx[i_a+lne]:-1;
12  idx_sh[lne]=(i_b+lne <
        l_b)?col_idx[i_b+lne]:-1;
13
14  __syncwarp();
15
16  // 2. Full comparison
17  for (int j = 0; j < WARP_SIZE; j++)
18   if ( my_idx == idx_sh[j] )
19     // perform the update and break
20  endfor
21
22  my_idx = __shfl_sync(my_idx, 31);
23
24  // move the head of the vectors for the next
       iteration
25  i_a += ((idx_sh[31]==-1) ||
26  (my_idx!=-1 && my_idx <= idx_sh[31]) ) *
       WARP_SIZE;
27  i_b += ((my_idx==-1) ||
28  (idx_sh[31]!=-1 && my_idx >= idx_sh[31]) ) *
       WARP_SIZE;
29  endwhile
```

could be a problem. Indeed, this strategy performs poorly in matrices that have many large rows. Therefore, it is reasonable to employ a different strategy for the row update stage in those cases. This implementation (ILU_{vect}) keeps the assignation of rows to $warps$. In this part, instead of working together to compute the intersection of the sparsity patterns as before, each thread of the warp takes one element of the current row and looks for an element in the reference row with the same column number using a binary search. While this strategy reduces the number of elements that are checked it also worsens the data locality (threads diverge and reads are not aligned) so this only makes sense when there are many elements to read. Thus we decided to keep the accumulation strategy of ILU_{shared} when both rows are small enough. After a preliminary testing we defined the use of the binary search update for cases where the current and reference rows have more than 128 non-zeros combined.

C. Adaptive variant (ILU_{adapt})

Since each row is processed by an entire $warp$, when there are less than 32 non-zeros in the current row some threads wait idle and computation power is lost. This is specially problematic in matrices with a high number of rows and only a few

Algorithm 4 Updating the current row using binary search

```
int i_a, int i_b // start of the vectors to intersect
int l_a, int l_b // length of the vectors
int * col_idx, volatile VALUE_TYPE * val // CSR sparse matrix
int lne // thread lane

for (int k = i_a+lne; k < l_a; k+=WARP_SIZE){

 int curr_col_idx=col_idx[k];
 beg=i_b+1; end=l_b-1;

 while(beg<=end)

  int med=(end+beg)>>1;
  int col_idx_med=col_idx[med];

  if(col_idx_med==curr_col_idx)
    // perform the update and exit
  else if(curr_col_idx > col_idx_med)
    beg = med+1;
  else
    end= med-1;
  endif

 endwhile
```

non-zeros per row. However, knowing which rows require less than 32 threads and assigning the correct number of threads to each row is not straightforward. We developed a new variant, which we call ILU_{adapt} which customizes the number of threads per row for each matrix depending of the average non-zeros per row. While this is not an optimal strategy, this feature can be calculated in $O(1)$ while reading the matrices. The synchronization and communication between threads working on the same row is done by defining *cooperative groups* [20] that subdivide the $warps$.

D. Level-set based variant

While the previous variants focused on optimizing the accumulation stage, the fourth one, which we call ILU_{levs}, aims to improve the strategy of row assignment to $warps$. The observation which motivates this variant is that, in sufficiently large matrices, dispatching row in numerical order may not be optimal since there can be rows ready to execute assigned to inactive blocks and warps, and active warps are more likely to busy wait a long time. To execute rows in a more efficient order we define a *level* structure.

To build the level structure we perform a BFS-like traversal in the graph corresponding to the lower triangular part of the matrix, assigning $lev(i)$ as the maximum level of the dependencies of i plus one. We use this to order the execution of the rows by dispatching all the rows of level i before the rows of level $i+1$. This reordering ensures that each row executes after all its dependencies and minimizes the waste of resources on rows that are are stuck busy-waiting.

V. Experimental results

This section presents the evaluation of our proposed routines and compares the result with `cusparse` and Intel MKL. In the rest of this section we use speedup of x against y as the division of their execution times (t_y/t_x). A speedup of more than 1 means that the version x is faster.

A. Reproducibility

The routines presented in this work are available at Dropbox [1] and will be uploaded to our GitHub [2] in the future. The tests are performed using a subset of 400 matrices from the SuiteSparse matrix collection [21]. All the the matrices in the test set are square matrices with at least 10000 rows. The list of matrices employed in the evaluation is also available in the GitHub repository.

The server used in the experimental evaluation has an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 64GB of RAM, 64kB of L1 cache, 256kB of L2 cache, and 8MB of L3 cache. The GPU employed is an Nvidia RTX 3090 Ti. The version of the CUDA Toolkit is 11.4. All experiments were performed using double-precision floating-point arithmetic.

B. Routines without preprocessing

In this section we present the results of our three routines that do not require a preprocessing stage. In this sense, we compare our three proposals (ILU_{shared}, ILU_{vect} and ILU_{adapt}) with `cusparse` and Intel MKL. We use the Intel MKL as the reference to normalize the results. Figure 4 presents the speedups regarding Intel MKL of the four routines (our routines and `cusparse` variant with `CUSPARSE_SOLVE_POLICY_NO_LEVEL`). We organized the results according to the number of levels of each matrix to study how the routines leverage the available parallelism in the matrix.

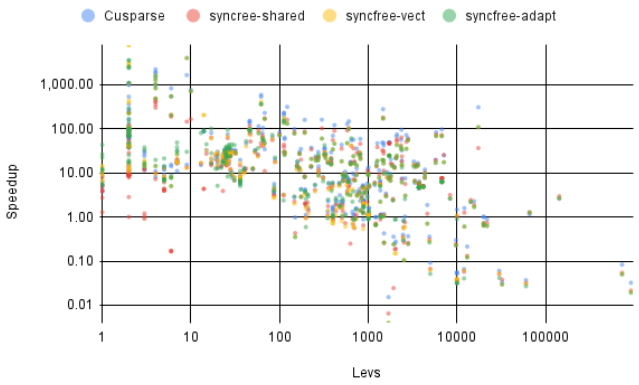

Fig. 4. Comparison against Intel MKL of the four routines: `cusparse`, ILU_{shared}, ILU_{vect} and ILU_{adapt}.

The results presented in Figure 4 show a similar behavior for all the routines, which worsens when the number of levels

[1] https://www.dropbox.com/scl/fi/n6ovo5hz62v01tq0qlwu8/sptrsv-ilu.zip?rlkey=sldvelvhefy77tq066nm926gy&st=2wriogrk&dl=0

[2] https://github.com/HCL-Fing/SPTRSV/tree/main

grows. This is reasonable since more levels usually imply less rows per level and more dependence between rows, giving less parallelism. In general, the worst results (speedups for all routines ≤ 0.1) are obtained in matrices with an average of rows per level between 1 and 2. An example of such behavior is the matrix `linverse` which has the same number of rows than levels (i.e. each row depends of the previous one) and for which all routines obtain a speedup between 0.04 and 0.07. In Table I we present the number of matrices in which each routine has worse performance than INTEL MKL and the median number of rows per level.

TABLE I
NUMBER OF MATRICES IN WHICH EACH ROUTINE IS OUTPERFORMED BY INTEL MKL AND MEDIAN OF THE NUMBER OF ROWS PER LEVEL OF THAT SET OF MATRICES.

	`cusparse`	ILU_{shared}	ILU_{vect}	ILU_{adapt}
N	27	47	45	41
$Med_{rows/lev}$	4.20	50.51	35.08	23.34

In Table I we show for each routine the number of cases in which INTEL MKL is faster. The results show that `cusparse` is the routine that does best compared with INTEL MKL, being outperformed in less matrices and only doing so when there is almost no parallelism possible. Comparing our three routines it is clear that while the number of matrices in which each variant is outperformed by INTEL MKL is similar, the median of $rows/lev$ decreases for ILU_{vect} and ILU_{adapt} regarding ILU_{shared}. This indicates that the latter are better at exploiting the available parallelism.

To conclude this section we compare our proposals with `cusparse`. In Table II we present the number of matrices in which each routine is the best of the group. For the purpose of this analysis we focus exclusively in the parallel implementations thus ignoring INTEL MKL. Following the previous reasoning about the number of levels, we focus on this metric for the analysis providing the breakdown for each quartile of number of levels.

TABLE II
NUMBER OF MATRICES IN WHICH EVERY ROUTINE IS THE BEST OF THE FOUR BOTH GLOBALLY AND BY QUARTILE OF NUMBER OF LEVELS.

	`cusparse`	ILU_{shared}	ILU_{vect}	ILU_{adapt}
Total	236	32	37	84
Q1	41	0	24	42
Q2	54	2	12	28
Q3	77	5	1	10
Q4	64	25	0	4

The results in Table II show that, while `cusparse` is superior in almost 2/3 of the matrices the results are not distributed evenly. For example in the first quartile (the quartile with less number of levels, and thus more parallelism) our routine ILU_{adapt} surpasses `cusparse` and ILU_{vect} wins in about 22% of cases against 38% of `cusparse`. The main advantage of `cusparse` is in the two last quartiles in which it obtains the best result in 83% and 68% of the matrices on each set. Finally, it is interesting to note that ILU_{shared} obtains its best results in the fourth quartile. It is important to note that in the contexts with more levels, using the `GPU` parallelism is not always justified since the INTEL MKL obtains better results in many matrices.

To graphically compare the performance of `cusparse` and our routines in contexts of different available parallelism, Figure 5 shows the speedups against `cusparse` ordered by the average number of rows per level instead of the number of levels.

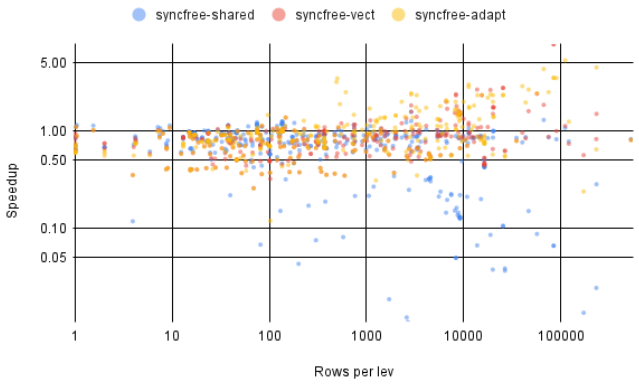

Fig. 5. Speedup of ILU_{shared}, ILU_{vect} and ILU_{adapt} against `cusparse`.

There are a few interesting observations about Figure 5. First, it is clear that in the matrices with enough parallelism (high average number of rows per level) the red and yellow points are generally above one and in many cases with considerable speedups. The second one is that ILU_{shared} (blue dots) have the inverse behavior, being competitive when there is not much parallelism but rapidly deteriorating as the number of rows per level growths. This suggest that ILU_{shared} is complementary to the other two versions. It would be interesting to explore an adaptive strategy to choose between the three implementations. This can be competitive with `cusparse` in the low parallelism environment while obtaining good speedups when the average number of rows per level growths. We plan to address this ins future work.

Finally, to give a general idea of the "competitiveness" of our routines, Table III presents the number of matrices for which our routines outperforms `cusparse` paired with the average speedup against it in the whole set. The results are encouraging because while ILU_{adapt} only wins in about 1/3 of the matrices,the average speedup is around 1. This suggests that with a better strategy to combine ILU_{vect} with ILU_{shared} (or INTEL MKL in the instances with low parallelism) we could have a variant that is not only competitive with `cusparse` as ILU_{adapt}, but ties or outperforms it in the majority of the matrices.

C. Routines with analysis

In this Section we present the results of our routine ILU_{levs} and compare them with those of `cusparse` using the flag

TABLE III
NUMBER OF MATRICES IN WHICH EACH ROUTINE OUTPERFORMS `cusparse` AND AVERAGE SPEEDUP AGAINST IT.

	ILU_{shared}	ILU_{vect}	ILU_{adapt}
N	49.00	76.00	113.00
$Avg_{speedup}$	0.74	0.90	1.01

`CUSPARSE_SOLVE_POLICY_USE_LEVEL`.As in the section above the results are presented as speedup against INTEL MKL.

Unlike with the *sptrsv*, the Incomplete LU factorization does not execute multiple iterations of the "solver" per execution of the preprocessing stage. In this line, Figure 6 presents the speedups of both `cusparse` and ILU_{levs} against INTEL MKL considering the execution time of the preprocessing and calculation stage combined.

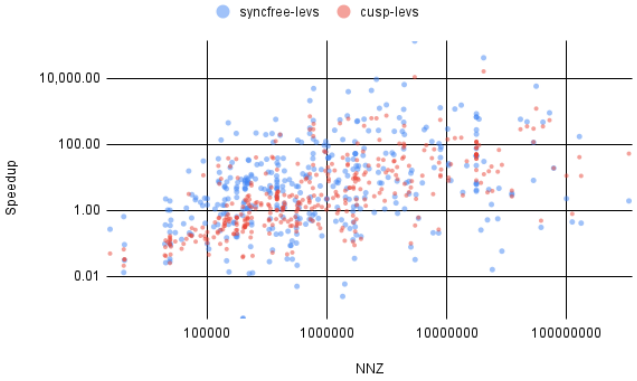

Fig. 6. Speedup of `cusparse` and ILU_{levs} against INTEL MKL as function of the number of non-zeros (NNZ).

The results show that, while both implementations outperform INTEL MKL in majority of the matrices, with enough non-zeros, our implementation generally outperforms `cusparse`. This is specially clear in the matrices with less than 10^6 non-zeros in which `cusparse` generally gets worse results than INTEL MKL. The impact is particularly important in the range of 10^5 to 10^6 non-zeros because while the implementation of `cusparse` obtains worse results than INTEL MKL, the new proposal surpasses it in the vast majority of matrices. To compare the two implementations more clearly, Figure 7 presents the speedup of our proposal against `cusparse`.

The results show, that the main improvements of our implementation are in the first two divisions (less than 10^6 non-zeros), reasonable speedups in the third one (between 10^6 and 10^7 non-zeros) and plateaus in the larger matrices with speedups between $0.5\times$ and $2\times$.

While the comparison with INTEL MKL must be done grouping the two stages together, it is interesting to compare `cusparse` against ILU_{levs} separating the two steps to asses the cost of the analysis and how it improves the solution stage. While we do not know how the `cusparse` versions are implemented, we assume that their preprocessing generates

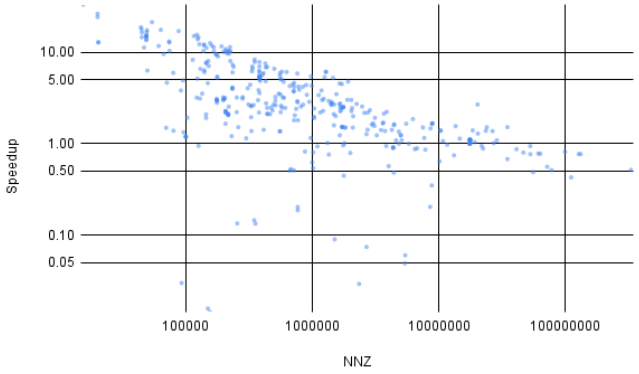

Fig. 7. Speedup of ILU_{levs} against `cusparse` as function of the number of non-zeros (NNZ).

a similar level structure to capture the dependencies. In this line, it is reasonable to compare the two stages one-to-one. This comparisons are presented in Figure 8.

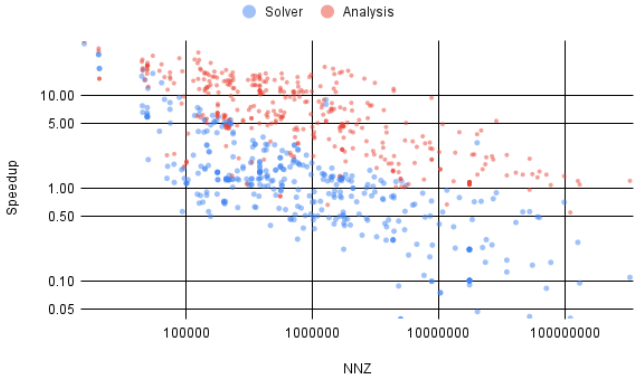

Fig. 8. Comparison between the `cusparse` routine using `CUSPARSE_SOLVE_POLICY_USE_LEVEL` and ILU_{levs} analysis (red) and solver (blue) stages as function of the number of non-zeros (NNZ).

The results show that our analysis outperforms `cusparse`'s in almost all matrices. However, the results of the computing stage are more mixed and after the 10^6 non-zeros threshold our routine gets worse results that `cusparse`. As before we expect that the two stages of `cusparse` are roughly similar to our implementation. In this sense, we think the analysis stage is a contribution in itself since it could be paired (or adapted for) the `cusparse` or other computing stage.

Finally, we think that it is interesting to study in which cases it is useful to use preprocessing and in which ones it is not. We present the comparison of both routines against the `cusparse` version that does not require preprocessing in Figure 9. The results show that while the preprocessing idea is only justifiable for `cusparse` in very large matrices (more than 50M non-zeros and mostly in the range $10-50$M) our version is competitive in the range $100000-1$M and gets better results with larger matrices. These results suggest that

our computation stage has significant room for improvement. Improving our computation stage and combining it with our faster level set analysis will yield an open source state-of-the-art routine for the ILU factorization.

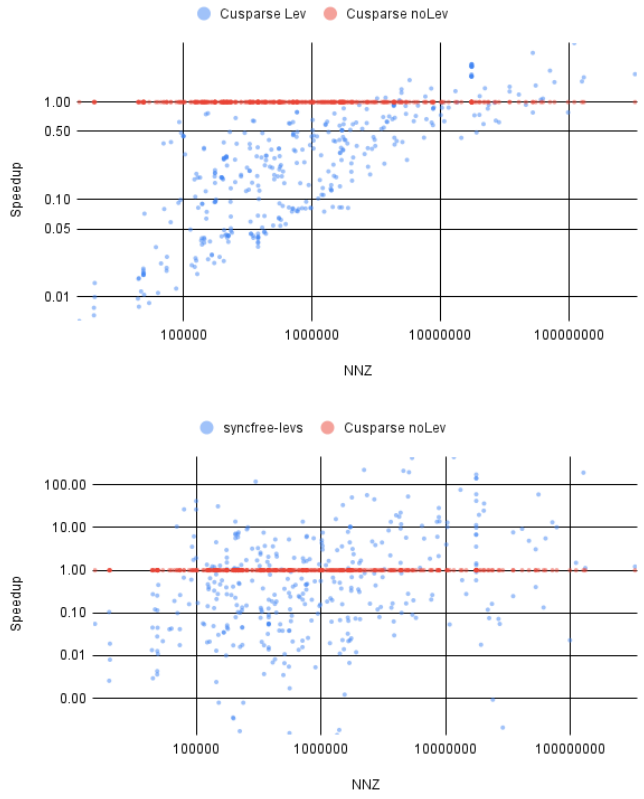

Fig. 9. Comparison between the `cusparse` routine using `CUSPARSE_SOLVE_POLICY_USE_LEVEL` (top) and our ILU_{levs} routine (bottom) against `cusparse` without preprocessing as function of the number of non-zeros (NNZ).

VI. Concluding remarks

This work studied the application of the synchronization-free methodology to the Incomplete LU factorization in GPUs. Furthermore, we delve into the method for computing the ILU-0, proposing alternative implementations for the most computationally demanding stage, i.e., the update of the row after the elimination of a coefficient. Although the overall performance is slightly below that of the principal vendor library (`cusparse`), our implementations of the update are competitive and show strengths for several matrix types. In this sense, improving the adaptive combination of these implementations is a promising line of future work. Finally, we provide a variant that leverages a level-set based preprocessing of the matrix. This variant outperforms `cusparse` in the majority of cases, mostly due to the superior performance of the analysis stage. These results indicate that slightly improving the computation stage can yield a performance clearly superior to the state of the art. In future work we will revisit the implementation of the update stage, with special focus on improving the data reuse. Additionally we wand to expand and deepen the evaluation focusing on both expanding the test set and the metrics evaluated in the routines.

Acknowledgment

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003

[2] T. F. Chan and H. A. Van der Vorst, *Approximate and Incomplete Factorizations*. Dordrecht: Springer Netherlands, 1997, pp. 167–202.

[3] D. Hysom and A. Pothen, "A scalable parallel algorithm for incomplete factor preconditioning," vol. 22, no. 6, 2001, pp. 2194–2215. [Online]. Available: https://doi.org/10.1137/S1064827500376193

[4] M. Naumov, "Parallel incomplete-lu and cholesky factorization in the preconditioned iterative methods on the gpu," *Nvidia Technical Report NVR-2012-003*, 2012.

[5] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the gpu," *NVIDIA Technical Report NVR-2015-001*, 2015.

[6] E. Chow and A. Patel, "Fine-grained parallel incomplete lu factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015. [Online]. Available: https://doi.org/10.1137/140968896

[7] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on gpus," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 1–16.

[8] A. George, M. T. Heath, J. Liu, and E. Ng, "Solution of sparse positive definite systems on a shared-memory multiprocessor," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 309–325, 1986.

[9] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *European Conference on Parallel Processing*. Springer, 2016, pp. 617–630.

[10] F. Zhang, J. Su, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "Yuenye-ungsptrsv: A thread-level and warp-level fusion synchronization-free sparse triangular solve," *IEEE Trans. Parallel Distributed Syst*, pp. 2321–2337, 2021.

[11] E. Dufrechou and P. Ezzatti, "Using analysis information in the synchronization-free GPU solution of sparse triangular systems," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 10, 2020. [Online]. Available: https://doi.org/10.1002/cpe.5499

[12] M. Freire, J. Ferrand, F. Seveso, E. Dufechou, and P. Ezzatti, "A gpu method for the analysis stage of the sptrsv kernel," *J. Supercomput.*, 2023. [Online]. Available: https://doi.org/10.1007/s11227-023-05238-8

[13] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[14] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurr. Comput. Pract. Exp*, vol. 29, no. 21, 2017.

[15] E. Dufrechou and P. Ezzatti, "Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm," in *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, I. Merelli, P. Liò, and I. V. Kotenko, Eds. IEEE Computer Society, 2018, pp. 196–203. [Online]. Available: https://doi.org/10.1109/PDP2018.2018.00034

[16] ——, "A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 2018, pp. 920–929. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00101

[17] L.-S. Chien, "How to avoid global synchronization by domino scheme," https://on-demand.gputechconf.com/gtc/2014/presentations/S4188-avoid-global-synchronization-domino-scheme.pdf, 2014, access date: 2023-20-06.

[18] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou, "Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 37–42.

[19] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using simd instructions." *ADMS@ VLDB*, vol. 1, no. 8, 2011.

[20] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda c++ programming guide: 11.7.0," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlwarp-reduce-functionsl

[21] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Nov. 2011.