A new level-set analysis and sparse storage format for the SPTRSV in GPUs

Manuel Freire, Ernesto Dufrechou and Pablo Ezzatti

***Abstract*—Due to its relevant role in many numerical methods, the solution of sparse triangular linear systems ($SpTRSV$) in parallel platforms is continuously studied to extract as much performance as possible from the latest hardware architectures. In the case of GPUs, the latest solvers use the synchronization-free paradigm. When the problem involves several system solutions for the same matrix, they often pre-process it through a level-set analysis to improve the equation solution scheduling in the solution phase. In addition, other optimizations address the load balancing issues and irregular memory access of the $SpTRSV$. In this work, we modify the classical approach to compute the level sets used in the parallel $SpTRSV$ computation, and we show that the new strategy generally reduces the computation time of the solver. Furthermore, we design an internal matrix representation that can significantly accelerate the solution stage at the cost of increasing the memory storage requirements of the algorithm. The experimental evaluation shows that the proposed modifications can improve the performance of a recent level-set and synchronization-free solver by up to 70%, significantly outperforming other state-of-the-art solvers, especially when several linear systems must be solved for each analysis phase.**

***Index Terms*—Sparse triangular linear systems, GPU, level-set analysis, synchronization-free methods**

I. Introduction

The solution of sparse triangular systems of linear equations (the $SpTRSV$ operation) is often the main computational bottleneck of many fundamental problems in science and engineering, such as solving general sparse linear systems or finding eigenvalues and eigenvectors [1].

Concretely, the $SpTRSV$ solves

$$Lx = b \qquad (\text{or} \quad Ux = b)$$

where L is a sparse lower triangular matrix (or U is sparse upper triangular), b is a dense vector, and x is the sought-after solution. The operation consists of the forward substitution (or backward) of the solved unknowns in the remaining equations. Unlike the dense case, where the parallelism in the solution is exploited through expressing the algorithm in terms of matrix multiplications and smaller triangular systems, the $SpTRSV$ leverages that substitutions are necessary only where the coefficient that multiplies the unknown is nonzero. This means that the equations depend only on a few previous ones, giving room to parallelization [2].

The data dependencies between the operations that compose the solution of different equations can be modeled using a directed acyclic graph (*DAG*). For example, in [3], the nodes in the graph represent multiplications, additions, and updates corresponding to different unknowns, and the edge (i, j) means that operation j must go after operation i. In [4], each node represents one row/column of the matrices. Analyzing the dependencies DAG allows determining the operations (or equations) that can be performed in parallel at a given discrete time. The set of equations that can be solved in parallel at a discrete time is often referred to as level-set [5]. If the level structure is known, the $SpTRSV$ can process the levels in sequence, solving the equations of each level concurrently.

The level-set is, therefore, a two-phase strategy. An important drawback of two-phase methods is that the analysis phase often takes considerably more time than the subsequent solution phase. However, in some scenarios, such as iterative methods, potentially many solution phases are executed for each analysis phase, so the high cost of the analysis is acceptable if it leads to a smarter parallel execution schedule and a smaller cost of the solution.

Another drawback of the level-set strategy is that, as each level is executed entirely before the next one begins, some equations can be unnecessarily stalled waiting for the previous level to finish, even when there are idle resources in the computing platform. In [6], the authors combine the level-set ordering of equations with synchronization-free solvers [7] so that rows of the next level that are ready to execute can start before the current level is completely processed. Moreover, they apply a warp partition strategy to avoid wasting GPU resources and achieve the accommodation of more equations in the GPU block wave. However, the computation of each equation's level is straightforward, meaning that the equation is solved as soon as possible, without considering if other equations depend on it. In this scheme, it can happen that equations on which no other depends delay the execution of equations on the critical path.

The main contribution of this work is a new $SpTRSV$ routine that addresses the principal shortcomings of the hybrid level-set/synchronization-free method in [6]. In particular, we propose two main optimizations:

- a new level-set based execution schedule that postpones the processing of rows not in the critical path.
- a new sparse format designed to improve the memory access pattern of the solution phase. This can lead to considerable runtime savings at the expense of an increase in storage.

The experimental evaluation, where we compare the new solver with [6] and other state-of-the-art publicly available libraries shows important speedups, especially for the matrices which take more time to compute.

The rest of this work is structured as follows. Section II presents basic concepts that provide context for the rest of the paper. Section III briefly surveys the state-of-the-art regarding the $SpTRSV$ implementation for GPUs. In Section IV, we describe our new level-set row ordering scheme. Section V presents the design of the new sparse storage format for the solution phase. In Section VI, we perform an extensive experimental evaluation and, finally, in Section VII, we present our conclusions and future work.

II. Background concepts

The $SpTRSV$ operation is challenging to parallelize because, unlike other sparse linear algebra kernels like the $SpMV$, it has inter-row dependencies and, potentially, every row i could depend on every other row j such that $j < i$. However, since the matrix is sparse, each row depends only on some previous rows determined by its nonzero coefficients.

The two principal approaches to parallelizing the $SpTRSV$ in GPU are level-set-based and self-scheduled (or synchronization-free methods). In the latter, the operations that perform in parallel are decided dynamically during the solution, while the former relies on a preprocessing stage to determine the execution schedule. Therefore, in level-set-based methods, the $SpTRSV$ is divided into two parts, the analysis stage, which is done only once per nonzero pattern, and the solving stage, different for every right-hand side.

Although the analysis stage can sometimes be costly, many applications of the $SpTRSV$ require the solution of several linear systems which share the coefficients matrix (or at least the sparsity pattern) and differ only in the right-side vector. In these situations, it can be advantageous to perform a heavy analysis to exploit deeper parallelism since the processing is reused.

A. Level-set scheduling

The level-set idea consists of organizing the equations (rows in the sparse matrix) into a sequence of sets. The equations in the first set do not depend on any other and can be solved immediately. Once the first set is processed, all the equations in the second set can be solved in parallel, and so on. The sets are constructed in the analysis stage, which calculates the rows on which each row depends. In the solving stage, all the rows that do not depend on any row at some point in the procedure can be processed, i.e., their unknowns can be substituted by the corresponding values. Once these equations are solved, the process repeats with those dependencies removed.

Classic implementations of the level-set scheduling include a barrier that prevents the processing of any equation of the next level until the current level is finished. A consequence of this is that rows that depend only on a few rows of the previous level have to wait for the entire level to finish. For example, in Figure 2, row 4 has to wait for rows 2 and 5 to finish because they are in lower levels, although it does not depend on them.

B. Synchronization-free scheduling

Another popular paradigm to solve sparse triangular linear systems is the synchronization-free (or self-scheduled) algorithms. This approach deals with the above synchronization problem by viewing the rows as a pool of tasks, each with a group of dependencies. When a thread is free, it takes a task from the shared pool and then polls for the dependencies. Secondary data structures are also needed to track which rows have been processed and the values of the solved unknowns.

Unlike the level-set-based approach, this paradigm avoids using a static order and thus does not require preprocessing the matrix. Additionally, it processes rows independently of their level. Continuing with the example of the matrix in Figure 1, row 4 only waits for row 3 to be processed (which waits for row 1) and does not care about rows 2 and 5. A description of this execution flow is presented in Figure 3.

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

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

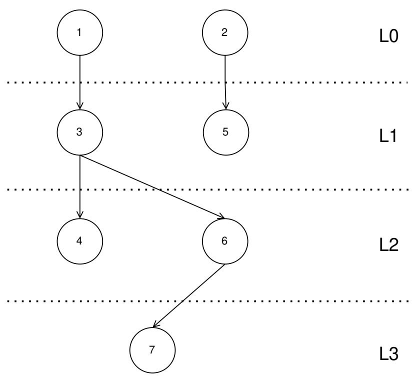

Fig. 2. Level structure generated from the example matrix using the level-set strategy.

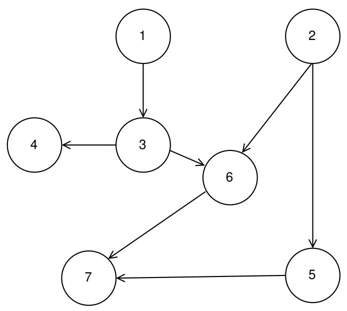

Fig. 3. Dependencies using the sync-free strategy for the matrix of Figure 1. Redundant dependencies were removed

III. Related Work

The first work in massively parallel $SpTRSV$ on GPU of the CUDA-era was done by M. Naumov [8]. He extended to GPU the ideas proposed for multi-cores by Anderson and Saad [9] in 1989 and later by Saltz [10]. This work focuses on situations where the same linear system has to be solved for a sequence of right-hand sides, such as iterative methods preconditioned by incomplete factorizations. The algorithm is divided into two steps: *analysis* and *solution*. To exploit the parallelism between rows that do not depend on each other, the analysis stage creates a level structure from the Directed Acyclic Graph (DAG) generated by the dependencies of the matrix L. The resolution stage solves the rows on the same level in parallel and synchronizes after finishing each level. This approach was incorporated by CUSPARSE. Other work in this line is [11] which used graph theory to divide the matrix into sub-matrices that fit in faster memories (e.g., the multiprocessor's shared memory). Finally, Suchoski et al. proposed applying graph coloring theory to find a permutation better suited for a level-set approach [12].

In 2016 Liu et al. observed that the synchronization was a big part of the computational cost of the previous approach, so they proposed an algorithm that avoided that cost [7]. They presented a routine that processes matrices in CSC format with a synchronization-free approach in which each non-zero is processed by one thread after their unknown is solved. The foundations of this approach come from the work of George et al. [13], who used this scheduling strategy on shared memory multiprocessors. The proposal was groundbreaking for the $SpTRSV$ operation and presented great performance results regarding CUSPARSE. Later they extended their proposal for multiple right-hand sides [14]. In [15], some of the same authors and others proposed a Sparse Level tile layout that allowed controlling the data reuse and built a parallel implementation aware of the data locality for the Sunway parallel processor.

In [16] the authors proposed a synchronization-free algorithm for CSR. Their solver avoids some of the performance pitfalls of [7], such as the extensive use of atomic operations. Instead of using one thread per non-zero, their routine assigns one $warp$ to each row, and each thread stores a partial sum before a final reduction stage. Unlike the previous efforts for GPU, their proposal does not need a preprocessing stage. In [17] the authors proposed a similar routine which they called ELMR.

The same authors from [16] later proposed a synchronization-free strategy to generate the level structure for level-set solvers [18], [19]. Finally, in [6], they combine the level-set and synchronization-free approaches into a new two-step routine. The first step of this routine, the analysis, uses their synchronization-free strategy to create a level structure to control the order in which rows are assigned to GPU warps. Furthermore, they use a $warp$ partition mechanism so that some warps can process more than one row in parallel. This proposal significantly improves the performance for matrices with many rows that have just a few nonzeros, since it can accommodate several of those rows in one $warp$, using processsors that otherwise would be idle during the entire execution. We present this routine in more detail in the next section since it is especially relevant to the current proposal.

In 2020 Su et al. built on the work of Liu et al. and proposed Capellini [20]. This solver works on CSR and processes each row in one thread. It iterates the row and busy waits when a dependency is not set. However, since a row could depend on others of the same $warp$ (which would cause deadlock in older GPU architectures) it does not busy wait for rows processed by the same $warp$ and after the main loop processes that dependencies sequentially. Later in [21] Zhang et al. improved the solver by adding a preprocessing stage that segments which rows will be processed in a thread or $warp$ level (i.e., if each $warp$ will process one or 32 rows).

Another line of work for the $SpTRSV$ is to divide the problem to exploit different sparsity patterns in the same matrix. In [22] the authors proposed a recursive blocked strategy that divides the matrix L into sub-matrices (squares and triangles) and applies $SpMV$ in the square parts exploiting the advantages of this kernel in parallel execution. The work done by Ahmad et al. in [23] is similar in the sense that they also use the approach of splitting the matrix but, instead of doing so based on a pre-selected recursion depth, it decides dynamically based on the characteristics of the matrix. Their implementation allows different execution policies (i.e., division of work) ranging from no split to split cross-platform (CPU-GPU) or in the same platform. The authors use a set of 657 matrices from the SuiteSparse [24] collection to train the model and a set of 327 for testing. They compare against Intel's MKL, ELMC, and CUSPARSE. This approach was also explored in CPU [25], [26].

If an exact result is not needed, an iterative approach can be studied. In [27] the authors explored the iterative solution of $SpTRSV$ on GPU, and in [28] Chow et al. worked with an iterative approach on the Incomplete LU factorization. Finally, an interesting line of work is proposed in [29]. The general idea is to use a hybrid approach (synchronization-free-level-set) to distribute the work between threads in a multi-core environment. While the work does not focus on GPU this idea could be explored in the context of GPU by limiting the number of threads to the maximum active threads.

A. Implementation details of [6]

This section provides some details on implementation of the $SpTRSV$ solver in [6] employ as the baseline for this work. We call SPTRSV$_{MR}$ for brevity. The source code is publicly available on the author's GitHub page[1].

SPTRSV$_{MR}$ is a two-stage routine. The solution phase of the routine partitions the $warp$ into equally-sized segments or *vectors*. Each vector processes one row of the matrix, and its size can be a power of two between 1 and 32. If the size of a vector is 2^p, the number of nonzeros of a row processed by that vector must be between $2^{p-1}+1$ and 2^p. To avoid deadlocks, the rows processed by a given $warp$ must be independent (which means they belong to the same level).

The mapping of rows to warps is determined during the analysis phase. First, a level-set analysis is performed to determine the level of each row. Then, for each level, the rows are sub-classified into seven bins according to their number of

[1]https://github.com/HCL-Fing/SPTRSV/tree/A-new-sparse-storage-format-and-level-set-analysis-for\-the-GPU-synchronization-free-SPTRSVFormat-Levels

nonzeros ($2^{p-1}+1 <= nnz <= 2^p$ for $p = 1, 2, 3, 4, 5, 6$ and the particular case where the partition is of size 1). After the classification, the rows are renumbered and distributed among the $warps$ such that the rows of each warp belong to the same bin.

The result of the analysis stage is a permutation vector of the rows, an array that limits the segment of the permutation vector processed by each $warp$, and an array with each $warp$'s partition size. During the solution phase, the threads of each partition fetch the coefficients from the corresponding row of the CSR matrix and enter the spinning stage, polling for the entry of the solution vector corresponding to their column index (initialized with *NaN*) until obtaining a valid value. After obtaining the values, the $warp$ multiplies them by the coefficient and accumulates the result in a local variable, which is later reduced using warp shuffle operations. The warp shuffle intrinsic allows performing several independent reductions, so the reduction of each partition is made in parallel.

IV. New level-set-based row ordering (SPTRSV$_{Lev}$)

Suppose we divide the solution into discrete times and solving any equation takes one time. Also, imagine that there is no limit to the number of equations that can be processed in parallel (at one time). In the level-set paradigm, the level of an equation i represents the first time in which we can accommodate its solution or, equivalently, the first time at which all of its dependencies are solved. Therefore, if j is the first unsolved equation containing a term with x_i, equation i can be solved anytime between times $level(i)$ and $level(j)$.

In an ideal situation like the above, where any number of equations can be solved at one time, there is no reason to delay the solution of an equation. However, in a scenario of limited computing resources, it may payoff to postpone the solution of an equation that is not in the critical path to bring forward the solution of one that is. Figure 4 shows an example of how the solution of a linear system can be accelerated by postponing the solution of equations that are not in the critical path.

As Section III-A explains, SPTRSV$_{MR}$ uses a level-set strategy to generate an ordering for the synchronization-free resolution stage. Particularly, it assigns rows to warps according to their level-set and number of nonzeros, meaning that all the rows in one level will be assigned to warps that start before the warps of the next level. Since GPUs can process only a finite number of rows in parallel, by assigning rows to warps this way, rows that are not part of the critical path can run earlier and leave those in the critical path waiting for processors.

We attacked the scheduling problem by changing how we assign rows to levels, sending rows not in the critical path to subsequent levels. In our new formulation, a level represents the last time a row can execute without delaying the execution of another dependent row. In other words, the level assigned to the row i is the maximum number that fulfills the following:

- it is higher than the maximum original level of i's dependencies (i.e., all the dependencies of row i are met)
- the maximum level of rows that depend (directly or indirectly) on i is lower or equal to the maximum level of the original level structure.

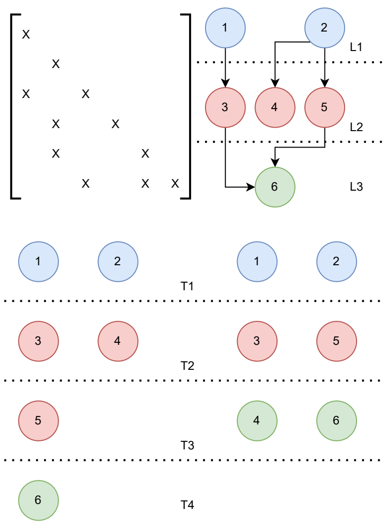

Fig. 4. Example of the solution of a linear system in a platform that can solve up to two equations in parallel. At the top, there is a sparse matrix and its original level structure. In the bottom left, equation 6 cannot be solved in time T3 because it depends on equation 5. In the bottom right, equation 4 is postponed and sent to the next level because it is not in the critical path.

We developed a version of the analysis stage of [6] using this new idea for the level structure. The level calculation is divided into two kernels: forward and backward iteration. The forward iteration works similarly to the previous kernel, starting from the first row and calculating the row level as the maximum of its dependencies levels plus one. The backward iteration starts from the leaves of the tree of dependencies. For the rows i that are not leaves in the tree, $level(i)$ is defined as the minimum level of all rows depending on row i minus one. Algorithms 1 and 2 show the pseudocode of both iterations.

Algorithm 1 Pseudocode of the forward iteration

```
for row = 1 to n in parallel do
  lvl = 0;
  // wait until dependencies are ready
  for col in row
    lvl = max(lvl,level(matrix[row,col]))
  end
  lvl_vect[row] = lvl+1
end
```

It is important to note that the backward iteration needs the number of rows depending on any given row (i.e., nnz by column). Since matrices are stored in CSR, this information is not easy to compute for a given row, so we added a preprocessing stage that counts, for each column, the number of non-zeros. This is stored in the `depending` vector.

In the backward iteration, each row waits until all the rows that depend on it have the final level assigned. Afterward, `lvl_vect` contains either the new row level (which was set by another thread) or the `MAXINT` constant. If it contains `MAXINT`, the corresponding entry of `lvl_vect` is updated with the maximum level number. After the new row level is obtained, the current row notifies the value to the rows waiting for it, which is equivalent to updating the entry of the

Algorithm 2 Pseudocode of the backward iteration

```
//lvl_vect is set to MAXINT between forward
    and backward iterations
for rows in parallel do
 while(depending[row] > 1){} // wait until
     dependencies of the row are ready
 if(lvl_vect[row] == MAXINT) //true if the
     row is leaf
  lvl = max_level
  lvl_vect[row] = lvl
 else
  lvl =lvl_vect[row]
 end
 for col in row
  lvl_vect[col] = atomicMin(lvl_vect[col],
      lvl-1)
  depending[col] =
      atomicSub(depending[col],1)
 end
end
```

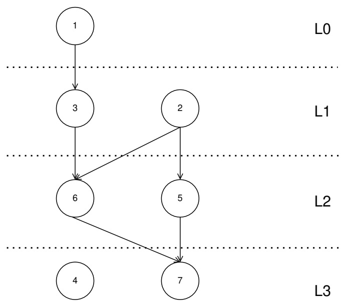

Fig. 5. Level structure resulting of the backward iteration

`level_vect` corresponding to those rows with the minimum between that entry and the row level minus one and decreasing by one their entry value in the `depending` vector (lines 10 to 13).

For example, the level structure for the matrix presented in Figure 1 is shown in Figure 5. The critical path (rows 1, 3, 6, and 7) is unchanged, but row 4 goes from level 2 to level 3 because no row depends on it. The edge from 3 to 4 disappears since it has no effect.

V. A SPARSE STORAGE FORMAT TO ACCELERATE THE SOLUTION PHASE (SPTRSV$_{Fmt}$)

As with many sparse GPU routines, SPTRSV$_{MR}$ suffers from irregular memory access patterns. For example, as the analysis phase delivers the execution schedule in the form of a permutation vector and not a physical reordering of the matrix, there is no guarantee that $warps$ from the same blocks will process contiguous rows, and thus it is difficult to exploit shared memory and take advantage of the caches. Moreover, grouping small rows into one $warp$ means that the $warp$ will fetch values of potentially several rows that are not stored contiguously in memory.

Specific sparse storage layouts are a classical strategy to improve the memory accesses of sparse routines such as the $SpMV$ or $SpTRSV$. Frequently, using a specific storage format implies the transformation of a matrix stored in a more general one, such as CSR, but generates significant savings if that matrix is reused for many operations.

We developed a new storage format that improves the memory access of the solution phase, borrowing ideas from CSR and ELL. A matrix in this format is represented by two arrays of $n_{wrp} \times 32$, where n_{wrp} is the required number of $warps$ in the solution stage (found in the analysis), plus an additional CSR-like structure for the rows that have more than 32 nonzeros. Each row of the $n_{warps} \times 32$ arrays holds the values and column indices, respectively, of the rows processed by each $warp$, so the $warps$ that process several small rows can fetch the matrix coefficients through a fully coalesced memory access. As in SPTRSV$_{MR}$, the $warps$ are subdivided into partitions of equal size, and the values and indices of each row are padded with zeros up to the next power of two.

For the $warps$ that process one large row, the first value in both arrays is used as a pointer to the CSR part of the structure that holds the row. As shown in Figure 6, the first element of each row of the column index matrix stores the value in the `row_ptr` vector on the CSR part of the structure (the index where the row starts in the `col_idx` and `values` vector). Also, in the same entry but of the values array, we store the index that points to the next row.

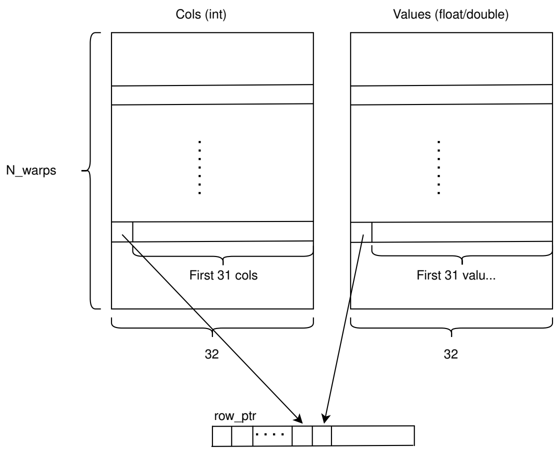

Fig. 6. Level structure generated from the example matrix using the level-set strategy.

It is easy to see that the format will have a more regular memory access pattern since reading the matrix coefficients will leverage the GPU coalesced loads even when the $warp$ processes a group of rows far away from each other in the original matrix. For large rows, the format adds one step of indirection, but at the same time, the index loading in the CSR structure also fetches the first 31 non-zeros of the row, which the warp can start processing right away. Additionally, the access to both matrices is aligned, something not guaranteed in CSR.

It is interesting to evaluate the memory cost of the new format and compare it with the cost of CSR. The cost of CSR can be calculated directly from the non-zeros and number of rows resulting in $nnz \times (4 + 8) + n + 1$ bytes where n is the number of rows and nnz the number of non-zeros. On the contrary, the cost of the new format concerns other variables like the length of the rows and the dependencies (which determine the number of $warps$ required). The ELL-like part of the format takes $32 \times (4+8) \times n_{wrps}$ bytes. On the other hand, the CSR-like part of the variable depends on the number of large rows and the number of non-zeros those rows have. It is challenging to analytically compare the cost of the two formats because they depend on different variables. However, it is easy to see that our format generally takes more memory since it uses the same number of bytes for the nonzeros plus the bytes dedicated to the zero padding.

VI. Experimental analysis

This section describes the experiments to validate our new routines and compares them against other publicly available libraries. We always calculate the speedup of routine X with regard to Y as the division of their execution times (t_Y/t_X). Therefore, speedup values higher than one mean that version X is faster. The evaluation is organized as follows. First, each contribution is evaluated separately, comparing the performance of the new routines with SPTRSV$_{MR}$. After that, the routines are compared with several publicly available state-of-the-art $SpTRSV$ libraries.

A. Experimental setup and reproducibility

All the routines presented in this work are available as Open-Source software at GitHub[2]. The tests are performed using the matrices of SuiteSparse matrix collection [24].

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 GPUs employed are an NVIDIA RTX 2080 Ti GPU and NVIDIA RTX 3090 TI. The version of the CUDA Toolkit is 11.4. All experiments were performed using double-precision floating-point arithmetic.

B. SPTRSV$_{Lev}$

We start the analysis of SPTRSV$_{Lev}$ by presenting its execution time next to the execution time of SPTRSV$_{MR}$. The results are presented in Figure 7. For visualization ease, we display the results ordered by the execution time of SPTRSV$_{MR}$ and divide them into three groups of matrices: those with execution time between 1 and 10 milliseconds, those between 10 and 100, and those higher than 100, leaving out the matrices that take less than a millisecond.

The first observation is that the difference in the execution time of the two variants is generally small. However, there are some matrices, especially in the top and middle figures, where the new strategy obtains a considerable performance advantage.

To understand the performance difference, we can analyze the results based on the levels since SPTRSV$_{Lev}$ modifies how the level sets are calculated. Figure 8 displays the speedup of SPTRSV$_{Lev}$ regarding SPTRSV$_{MR}$ as a function of the number of levels.

[2]The reference to our GitHub page has been omitted for the blind review process but will be included in the final version.

As a general trend, we can see that when the number of levels grows, the speedups below one are less frequent. When the matrix has few levels, there are fewer restrictions to the parallelism, and, probably, rows execute mostly following the original order of the matrix. Additionally, keeping the original order allows a better memory locality because the rows processed by each warp are more likely to be contiguous in the matrix CSR data structure, and moving some rows to another level set (and hence another warp) can have a negative impact. For these reasons, it is reasonable to expect little advantage in those cases.

When the number of levels grows, there is room for changes with more impact (there can be more rows outside the critical path that can move forward to another level). Moreover, the reorganization performed by SPTRSV$_{Lev}$ is more aggressive, and the memory access is already distorted. However, in matrices with many levels (for example, tridiagonal matrices) it is likely that there are not many changes to be done to the level set structure, so the execution time is almost the same for the two variants.

In this line, we explored several metrics to quantify the difference between the traditional and the new level structure. These metrics are the maximum level change for a row, the number of rows that are not affected (i.e., the number of rows in which the change is 0), and the average change for all rows. Preliminary studies showed that the first two metrics are closely related to the number of levels. Figure 9 shows the acceleration results sorted by the average change of the row levels.

There are a few observations from this graph. The average is a good estimate of how much the level structure changed. Therefore, it is reasonable that when this is low, the results are similar to the baseline. When the average is between 1 and 100, the results are mixed, with an advantage of up to 40% for one variant or the other. Finally, when the average is more than 100, the speedups are generally above one. A reasonable conclusion is that when the change is enough, it positively affects the overall running time by prioritizing unknowns that will be needed first. Conversely, when the change is moderate, the gains obtained by the smarter execution scheme can be outweighed by a worse use of data locality and/or cache coming from separating contiguous rows.

Considering the above, the solver can incorporate this optimization, and we expect the positive results will outweigh the bad ones. However, it is interesting to explore an adaptive approach that allows choosing dynamically between the two strategies. As the analysis phase of the SPTRSV$_{Lev}$ variant includes the one from the SPTRSV$_{MR}$, it is possible to decide whether to continue with the second part as soon as the first is finished. The number of levels obtained after the computation of the first level structure is the leading candidate to perform such a choice. Such approach would need to prioritize to make the right decision in the matrices that take more time.

C. SPTRSV$_{Fmt}$

As in the previous section, we start by presenting the execution times of the SPTRSV$_{Fmt}$ routine compared to

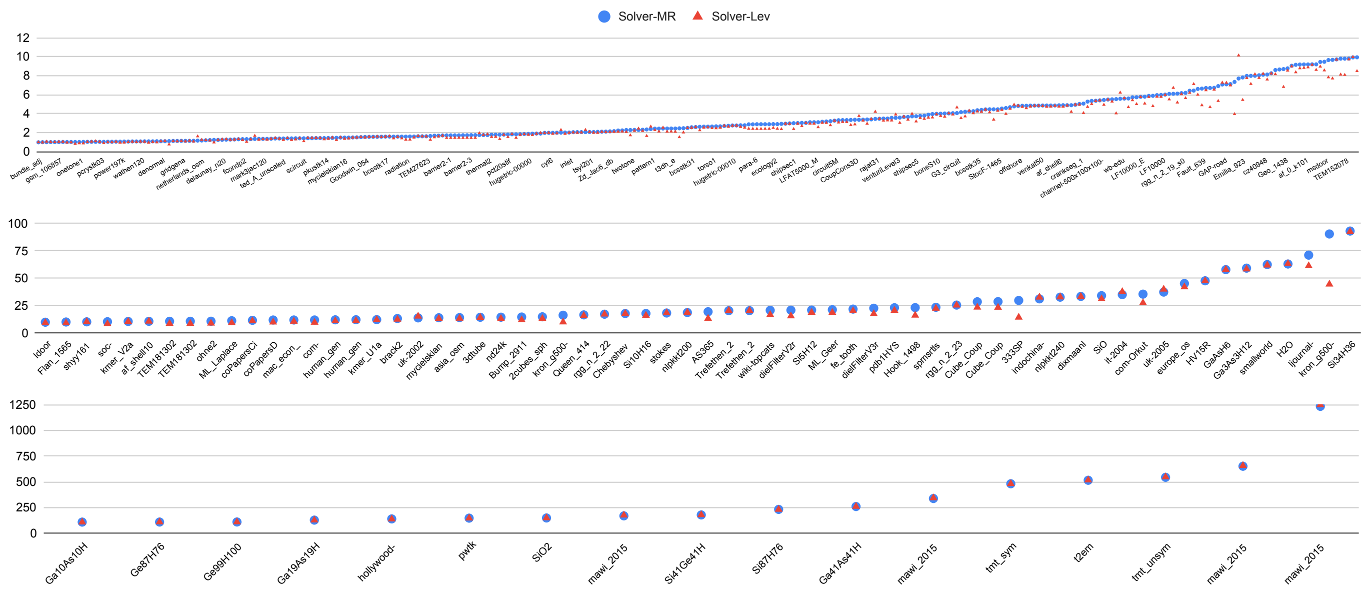

Fig. 7. Execution time of SPTRSV$_{Lev}$ and SPTRSV$_{MR}$ (in milliseconds). The results are ordered by the execution time of SPTRSV$_{MR}$ and divided into three groups of matrices: those with execution time between 1 and 10 milliseconds (top), those between 10 and 100 (middle), and those higher than 100 (bottom). Matrices that take less than a millisecond are left out.

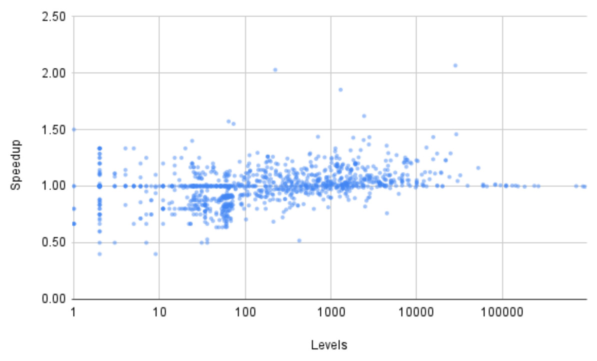

Fig. 8. Speedup of SPTRSV$_{Lev}$ against SPTRSV$_{MR}$ as a function of the number of levels.

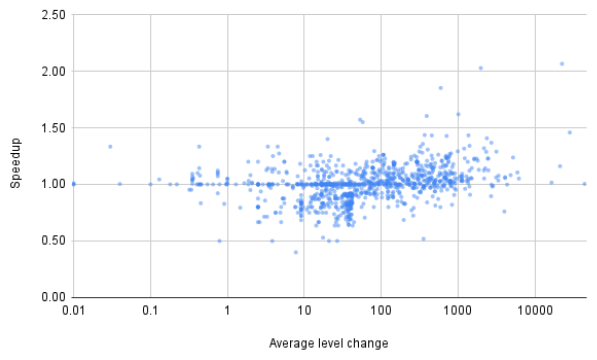

Fig. 9. Acceleration obtained by SPTRSV$_{Lev}$ regarding SPTRSV$_{MR}$ as a function the average change in the matrix level set structure.

those of SPTRSV$_{MR}$. Figure 10 shows a similar pattern to that of the SPTRSV$_{Lev}$ variant, where the best accelerations are obtained for matrices that take between 1 and 10 ms. Moreover, it is worth noting that our proposal outperforms or at least equals the baseline results for most cases. Unlike before, the performance of this variant does not present an obvious relation with a specific metric. Figure 11 shows the speedup of SPTRSV$_{Fmt}$ against nnz of the matrix.

The general tendency is that when the matrices are larger, the results improve with speedups up to $1,77\times$. We note a considerable cluster of matrices of 64.000 to 85.000 nonzeros where the accelerations range from $0,5$ to $1\times$. A detailed analysis shows that the cluster mostly comprises a group of matrices called `as_caida_G_...` where the new format performs poorly against the baseline. As expressed earlier, a relevant aspect of this routine is the memory overhead introduced by the new structure. Figure 12 shows the memory usage of CSR and the secondary structure. The areas represent the logarithm of the memory used in Bytes independently of each other (i.e., the red area does not incorporate the blue area). Generally, the secondary structure uses similar amounts of memory than the original CSR representation, meaning that the memory cost of the new solver is $2\times$ the baseline.

To conclude the analysis, we explore the trade-off between increasing the execution time of the analysis stage and the improvement in the solver in the scenario of an iterative solver, where several solution stages are performed for each analysis, and the savings in the solution phase can compensate for the cost of such analysis. Figure 13 shows the iterations required to compensate for the analysis overhead produced by SPTRSV$_{Fmt}$ against SPTRSV$_{MR}$ while Figure 14 presents the same comparisons for SPTRSV$_{FL}$ (new format plus new level structure) against the two mentioned before. We did not include the points for cases where the baseline outperforms the new solver. For Figure 13, this happens for 209 matrices, while for Figure 14, there are 475 and 300 against SPTRSV$_{Fmt}$

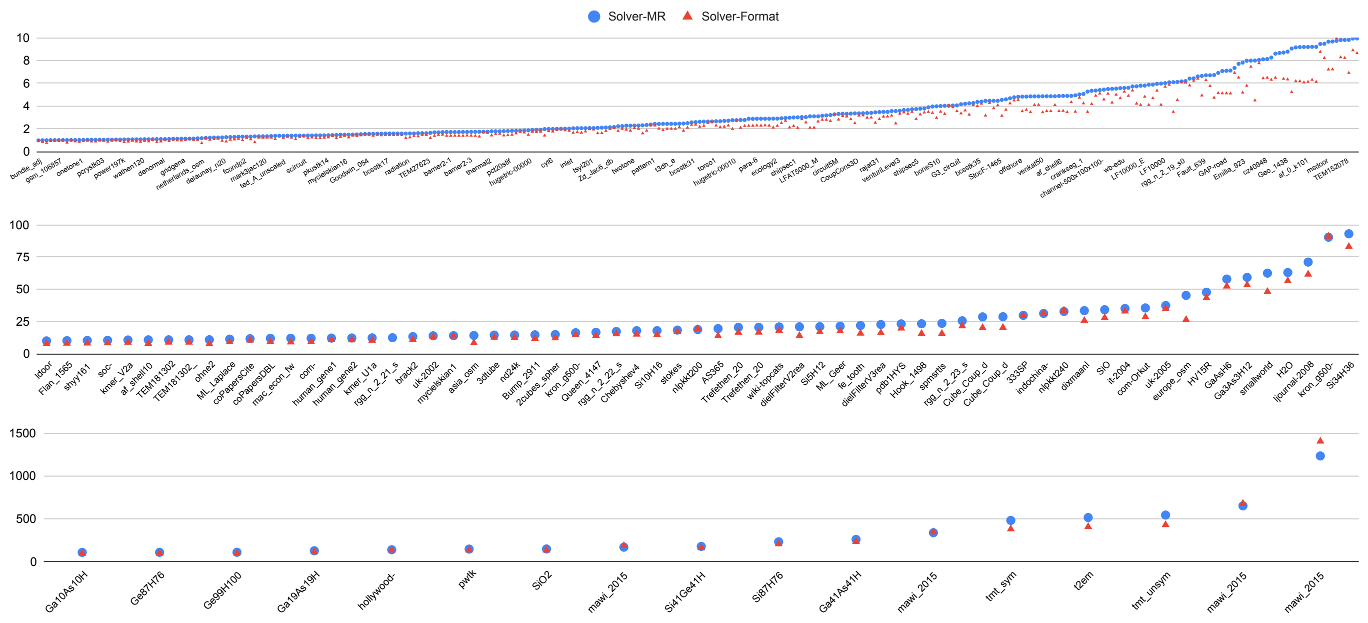

Fig. 10. Execution time of SPTRSV$_{Fmt}$ and SPTRSV$_{MR}$ (in milliseconds). The results are ordered by the execution time of SPTRSV$_{MR}$ and divided into three groups of matrices: those with execution time between 1 and 10 milliseconds (top), those between 10 and 100 (middle), and those higher than 100 (bottom). Matrices that take less than a millisecond are left out.

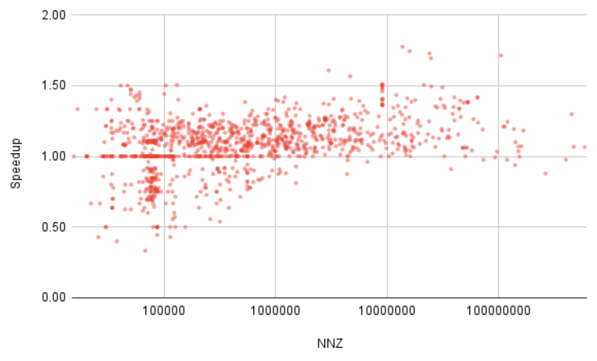

Fig. 11. Speedup of SPTRSV$_{Fmt}$ against SPTRSV$_{MR}$ as a function of the matrix size (as the number of non-zeros).

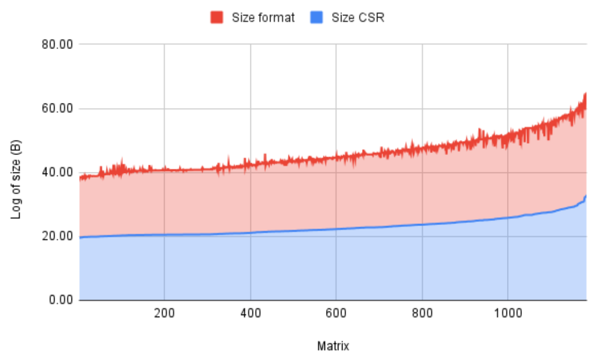

Fig. 12. Log of the memory cost (on Bytes) of the CSR structure and the secondary structure.

and SPTRSV$_{MR}$ respectively. The number of matrices in the test set is 1182.

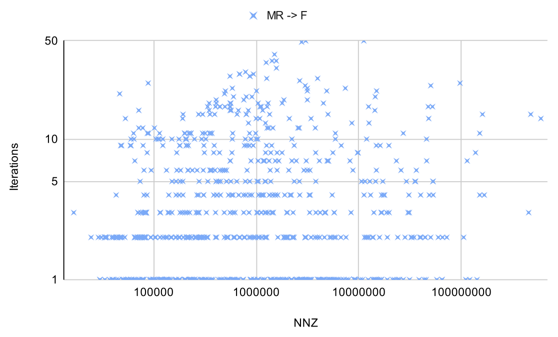

Fig. 13. Number of iterations required to justify the execution of the analysis of SPTRSV$_{Fmt}$.

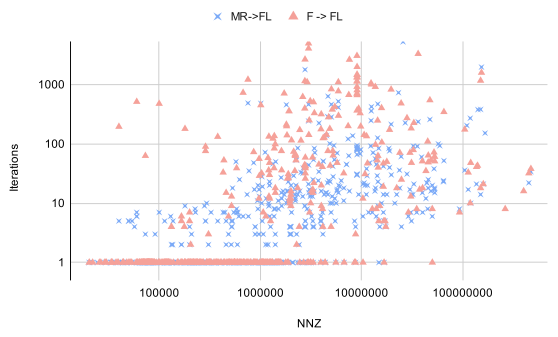

Fig. 14. Number of iterations required to justify the execution of the analysis of SPTRSV$_{FL}$. This graph shows all the matrices in which the solver of this routine outperforms either the one of SPTRSV$_{MR}$ or SPTRSV$_{Fmt}$.

The results show that, in most matrices, the solver of

SPTRSV$_{Fmt}$ requires 10 iterations or less to outweigh the analysis. It is important to note that there are many matrices in which, with only one iteration, we compensate for the analysis cost. On the other hand, SPTRSV$_{FL}$ needs more iterations (up to 100) to equate to SPTRSV$_{MR}$ (and generally more to catch up to SPTRSV$_{Fmt}$).

D. Comparison with the state of the art

To conclude the experimental evaluation of our proposal, we compare it with other recent GPU implementations of the $SpTRSV$ for GPU. In particular, we chose four well-known publicly available libraries: CUSPARSE [30], SYNCFREE [31], CAPELLINI [20] and YUENYEUNG [21].

Although CUSPARSE's may not be the best implementation of the $SpTRSV$, it is a mature library that has been maintained and revised through several years. We compare our results with the two-phased `cusparse_csrsv2` routine (with the flag `CUSPARSE_POLICY_LEVEL`), which is comparable to our routine and presumably uses a self-scheduled synchronization [32]. The other implementations were presented in recent works and are available in the GitHub platform[3].

Our routines involve a somewhat heavy analysis phase and are intended for iterative solvers where many iterations of the solution phase are required for each analysis. Although our new analysis routine is not fully optimized, we include the time of the analysis to be fair with CAPELLINI and YUENYEUNG, which only involve a lightweight preprocessing stage. We present three different comparisons. Figure 15 shows the execution time of the analysis plus 100 iterations of the solver for the four routines mentioned above. Second, in Figure 16 we present a scenario with only 10 solver iterations plus analysis. Finally, in Figure 17, we focus only on the solver.

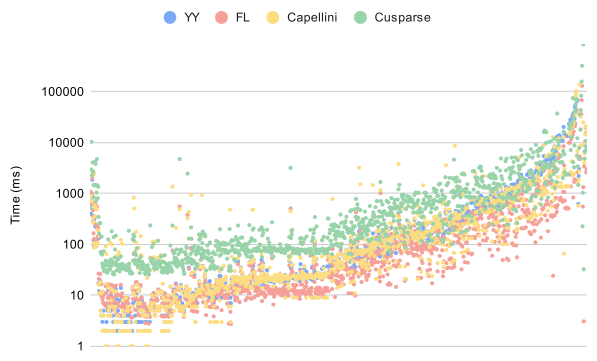

Fig. 15. Comparison of the execution time of 100 iterations of the solver plus analysis for our proposal and other publicly available libraries, CUSPARSE, CAPELLINI and YUENYEUNG.

Figure 15 indicates that CUSPARSE is the worst routine in almost all matrices and by a considerable difference. Between CAPELLINI and YUENYEUNG the results are fairly similar, and finally, our proposal is the one with the best results in most of the matrices. More precisely, the new proposal outperformed all the others in 922 out of 1156 matrices for which all four routines finished the execution successfully.

[3]https://github.com/JiyaSu/CapelliniSpTRSV

Since Figure 16 represents a context with only a few iterations per matrix, we also include SPTRSV$_{Fmt}$ since its analysis stage requires sensibly less time. Unlike in the previous analysis, our routines have the best results in 541 matrices, whereas CAPELLINI or YUENYEUNG routines outperform ours in 615. However, our proposals get better results when the execution time is higher.

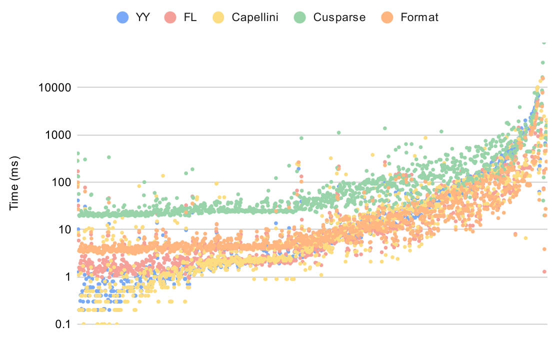

Fig. 16. Comparison of the execution time of 10 iterations of the solver plus analysis for our proposal and other publicly available libraries, CUSPARSE, CAPELLINI, and YUENYEUNG.

Finally, we repeat the analysis focusing only on the solver. Figure 17 is similar to Figure 15, but the number of matrices for which our routine has the best results grows to 1000.

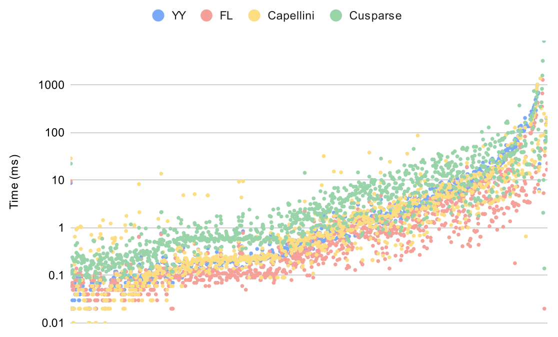

Fig. 17. Comparison of the execution time of the solver stage of our proposal and other publicly available libraries, CUSPARSE, CAPELLINI and YUENYEUNG.

To show the results more clearly, we present a summary in Table I. This table presents the median, interquartile range (IQR), minimum, and maximum speedup against CUSPARSE for 100, 10, and 1 iteration. The values in dark and light gray show the best and second-best routines. In the scenario with high iterations (100 plus analysis) and when we consider only the solver, our routines outperform YUENYEUNG and CAPELLINI. In contrast, if we consider a scenario with only a few iterations (10 plus analysis), we get mixed results with our proposals having a higher floor but a lower ceiling. This results are expanded by presenting the boxplots in Figure 18.

The results presented in Figure 18 show a similar picture to Table I in which our routines have the results concentrated in a smaller range. In contrast, the other two routines have a higher variance. Moreover, both in CAPELLINI and YUENYEUNG,

		YY	Cap	FL	Fmt
1	Min	0.07	0.07	1.25	1.14
	Median$_{IQR}$	$2.88_{1.76}$	$3.03_{1.99}$	$5.33_{2.29}$	$5.37_{2.26}$
	Max	22.41	88.5	126.36	110.95
10	Min	0.19	0.13	0.6	1.33
	Median$_{IQR}$	$8.25_{16.75}$	$8.07_{16.01}$	$8.31_{11.63}$	$5.75_{6.37}$
	Max	102.8	205.6	57.95	51.65
100	Min	0.08	0.08	1.25	1.26
	Median$_{IQR}$	$3.67_{2.65}$	$3.64_{2.52}$	$6.06_{2.09}$	$5.54_{1.87}$
	Max	26.36	91.36	110.84	97.65

TABLE I
SUMMARY OF THE SPEEDUPS AGAINST CUSPARSE OF THE FOUR ROUTINES.

there is a more significant number of values outside the right whisker (i.e., more than $Q3 \times 1.5 \times IQR$) and with the maximum speedups surpassing the three figures. However, it is important to note that the speedups are agnostic to the overall execution time of the routines, and, as we noted before, our routines outperform YUENYEUNG and CAPELLINI in the matrices that take more time. In Table II we compare the global time (and speedup against CUSPARSE) on the whole set.

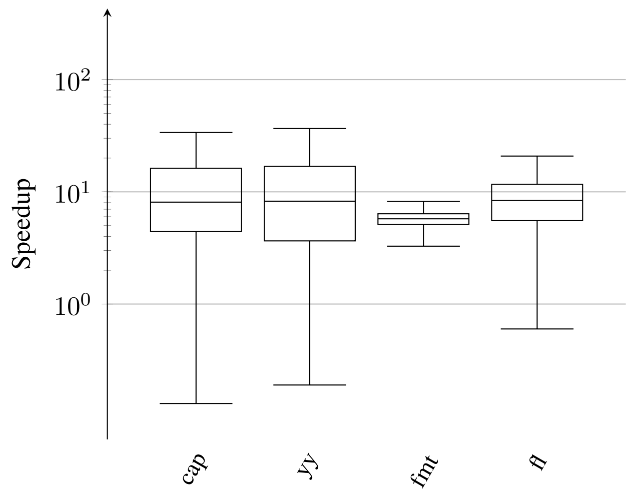

Fig. 18. Boxplots for the distribution of speedups against CUSPARSE for SPTRSV$_{Fmt}$, CAPELLINI, YUENYEUNG and SPTRSV$_{FL}$.

	Total time (ms)	Speedup
Cusp	296,328.80	1.00
Fmt	94,367.60	3.14
FL	111,899.20	2.65
Cap	122,792.00	2.41
YY	172,731.61	1.72

TABLE II
AGGREGATED EXECUTION TIME OF EACH ROUTINE FOR ALL MATRICES, AND SPEEDUP AGAINST CUSPARSE CONSIDERING THE AGGREGATED TIME, IN THE 10 ITERATIONS PLUS ANALYSIS SCENARIO.

The results shown in Table II present a different picture, with SPTRSV$_{Fmt}$ being the best routine. This is expected since the gains in the solver of SPTRSV$_{FL}$ do not compensate for the analysis cost in this number of iterations. The same behavior is shown between CAPELLINI and YUENYEUNG in which the advantage of the former can be explained by having no analysis. Finally, our routines outperform all the others.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we proposed a new two-phase implementation of the $SpTRSV$ routine. This routine uses an improved storage format and a new level-set scheme to optimize the row order.

We compared our routine against different publicly available implementations. The results show that the new solver is faster than the state-of-the-art routines in the scenario where the result of the analysis can be reused in subsequent solution phases.

Among the possible lines of future work, we consider applying these ideas to other sparse linear algebra operations. Second, we are interested in expanding our analysis routine using an adaptive approach that, depending on the problem, decides which level-set strategy to adopt. Finally, we will optimize the implementation of the analysis stage, specifically the generation of the new level-set structure.

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] J. Mayer, "Parallel algorithms for solving linear systems with sparse triangular matrices," *Computing*, vol. 86, no. 4, pp. 291–312, 2009.

[3] O. Wing and J. W. Huang, "A computation model of parallel solution of linear equations," *IEEE Trans. Computers*, vol. 29, no. 7, pp. 632–638, 1980. [Online]. Available: https://doi.org/10.1109/TC.1980.1675634

[4] J. Saltz, *Automated Problem Scheduling and Reduction of Synchronization Delay Effects*, ser. ICASE report. NASA Langley Research Center, 1987. [Online]. Available: https://books.google.com.uy/books?id=QqdBAQAAMAAJ

[5] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, pp. 856–869, Jul. 1986. [Online]. Available: http://dx.doi.org/10.1137/0907058

[6] 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

[7] 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.

[8] 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.

[9] Anderson, E. and Saad, Y., "Solving sparse triangular linear systems on parallel computers," *Internat. J. High Speed Comput.*, vol. 1, pp. 73–95, 1989.

[10] Saltz, J., "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. and Stat. Comput.*, vol. 11, pp. 123–144, 1990.

[11] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides, "Balancing locality and concurrency: Solving sparse triangular systems on gpus," in *HiPC*. IEEE Computer Society, 2016, pp. 183–192.

[12] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting sparse triangular solution to gpus," in *ICPP Workshops*. IEEE Computer Society, 2012, pp. 140–148.

[13] 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.

[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] X. Wang, W. L. 0002, W. Xue, and L. Wu, "swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. ACM, 2018, pp. 338–353.

[16] 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

[17] R. Li and C. Zhang, *Efficient Parallel Implementations of Sparse Triangular Solves for GPU Architectures*, pp. 106–117. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.10

[18] E. Dufrechou and P. Ezzatti, “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

[19] 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

[20] J. Su, F. Zhang, W. Liu, B. He, R. Wu, X. Du, and R. Wang, “CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs,” *Proceedings of the 49th International Conference on Parallel Processing*, 2020.

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

[22] Z. Lu, Y. Niu, and W. L. 0002, “Efficient block algorithms for parallel sparse triangular solve,” in *ICPP*. ACM, 2020, pp. 63:1–63:11.

[23] N. Ahmad, B. Yilmaz, and D. Unat, “A split execution model for sptrsv,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2809–2822, 2021.

[24] 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.

[25] A. M. Bradley, “A hybrid multithreaded direct sparse triangular solver,” in *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing (CSC)*. SIAM, 2016, pp. 13–22.

[26] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, “Sparsifying synchronization for high-performance shared-memory sparse triangular solver,” in *ISC*, vol. 8488, 2014, pp. 124–140.

[27] H. Anzt, E. Chow, and J. J. Dongarra, “Iterative sparse triangular solves for preconditioning,” in *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9233, 2015, pp. 650–661.

[28] E. Chow and A. Patel, “Fine-grained parallel incomplete lu factorization,” *SIAM J. Sci. Comput*, vol. 37, no. 2, 2015.

[29] P. Sandhu, C. Verbrugge, and L. J. Hendren, “A hybrid synchronization mechanism for parallel sparse triangular solve,” in *Languages and Compilers for Parallel Computing - 34th International Workshop, LCPC 2021, Newark, DE, USA, October 13-14, 2021*, vol. 13181, 2021, pp. 118–133.

[30] “cusparse, the cuda sparse matrix library,” https://docs.nvidia.com/cuda/cusparse/index.html, access date: 2023-15-06.

[31] W. Liu, A. Li, I. S. D. Jonathan D. Hogg, and B. Vinter, “A synchronization-free algorithm for parallel sparse triangular solves,” *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings, Lecture Notes in Computer Science*, vol. 9833, pp. 617–630, 2016.

[32] 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.