Graph Transformation and Specialized Code Generation For Sparse Triangular Solve (SpTRSV)

Buse Yılmaz
Computer Science
İstinye University
İstanbul, Turkey
buse.yilmaz@istinye.edu.tr

***Abstract*—Sparse Triangular Solve (SpTRSV) is an important computational kernel used in the solution of sparse linear algebra systems in many scientific and engineering applications. It is difficult to parallelize SpTRSV in today's architectures. The limited parallelism due to the dependencies between calculations and the irregular nature of the computations require an effective load balancing and synchronization mechanism approach. In this work, we present a novel graph transformation method where the equation representing a row is rewritten to break the dependencies. Using this approach, we propose a dependency graph transformation and code generation framework that increases the parallelism of the parts of a sparse matrix where it is scarce, reducing the need for synchronization points. In addition, the proposed framework generates specialized code for the transformed dependency graph on CPUs using domain-specific optimizations.** [1]

***Index Terms*—SpTRSV, graph transformation, specialized code generation**

I. Introduction

From earth sciences to physics and chemistry, sparse linear systems are found in several applications of science and engineering such as computational fluid dynamics, reservoir simulation and finite element modeling. SpTRSV is the building block for several numerical solutions and it consumes a significant portion of the total execution time. Direct and iterative methods for sparse linear algebra systems and sparse matrix factorizations such as LU, QR and Cholesky are examples to numerical solutions that use SpTRSV. Parallelizing SpTRSV has been studied extensively and yet it remains a challenging problem even with today's highly parallel and specialized architectures, which deliver a tremendous amount of computation per second. The main challenges are: **1)** SpTRSV exhibits limited parallelism due to the dependencies between computations, **2)** Computations are irregular: workloads assigned to threads are fine-grained and vary in granularity due to the sparsity structure of the matrix, **3)** Challenges 1 and 2 result in significant synchronization and management overhead, **4)** Computational dependencies require a careful analysis and an efficient load balancing approach is needed due to the fine-grained workloads.

The challenges arise from the nature of SpTRSV together with the sparsity pattern of the matrix: the sparsity pattern is usually non-uniform throughout the matrix. Therefore, a sparse matrix has parts exhibiting different degrees of parallelism. SpTRSV cannot benefit from today's highly parallel architectures for the parts that exhibit very few parallelism. These parts usually have few rows, therefore, several cores sit idle due to the dependencies.

To remedy the challenges mentioned above, we propose a dependency graph transformation and code generation framework to:

- Make the sparsity pattern of the matrix more uniform by increasing the parallelism where it is scarce
- Reduce the need for synchronization points
- Generate specialized code for SpTRSV on CPUs, using domain-specific optimizations

To increase the degree of parallelism in parts of the matrix where the computation becomes serial, we propose to alter the sparsity pattern by transforming the dependency graph by rewriting the equations rows represent. Rewriting the rows enables breaking the rows' dependencies in a safe way, hence these rows can be moved to upper levels when level-set approach [2], [18], [19] is used. Generating specialized code for SpTRSV enables several optimizations such as reducing the memory accesses and indirect indexing, and applying arithmetic optimizations.

The rest of the paper is as follows: Section II gives background information about SpTRSV and Section III introduces the equation rewriting approach. Matrix analysis and specialized code generation is presented in Section IV, the following section, Section V, presents the experiment results and we conclude in Section VI.

II. Background

The sparse triangular solve takes a lower triangular matrix L and a right-hand side vector b and, it solves the linear equation $Lx = b$ for x.

Figure 1 presents the operation $Lx = b$ together with the dependency graph of the lower triangular matrix L partitioned into levels on the right. Dependency graph of L is a directed acyclic graph (DAG_L), where the nodes represent the rows

[1]This paper is a revised version of the paper appeared in http://www.basarim.org.tr/2020/doku.php with the name "Buse Yilmaz and Didem Unat. *Seyrek Alt Üçgen Matris Çözümü için Graf Dönüşümü ve Özelleşmiş Kod Üretimi*". The presentation in English can be found at (starts at min 10) https://www.youtube.com/watch?v=fX2yM_gNHj8&list=PLRofGbhRbm7j3LaxVB3K_7hwOs1-1Serc&index=12&ab_channel=TUBITAKULAKBIMTV

and the edges represent the dependencies between nodes. Algorithm 1 on the right handside presents a serial forward substitution implementation of SpTRSV for solving $Lz = b$. n represents the number of rows in L. For each row, a partial sum is calculated using the nonzeros (dependencies) in that row in the inner for loop. The inner loop can be parallelized for each row and the outer for loop can be parallelized to the extend that dependencies permit. Unless all dependencies are met, a row cannot be calculated. This causes a bottleneck on the parallelization of outer for-loop.

The literature is rich in approaches to optimize SpTRSV: Level-set methods [2], [18], [19] group the rows that have no dependencies to each other into levels or wavefronts . The rows in a level are executed in parallel and the levels are executed one after the other (serially). In [18], a level-set based method for SpTRSV is presented for CPUs where matrix reordering is used for the coefficient matrix to address the performance problem caused by poor spatial locality of the data. Authors advocate for dynamic scheduling performing worse due to the fine-grained nature of the computations in SpTRSV. Later, this approach is adopted for NVIDIA GPUs in [14] and in [10]. A recent work extended the level scheduling for Sunway architecture [23]. The authors propose a new data layout, named sparse level tile (SLT) format, which improves data locality.

Synchronization-free methods emerged as an alternative to level-set methods to eliminate the need for building level sets, and hence the need for barriers at the end of each level. First examples of this approach were on CPUs in [6]. Later, several implementations on GPUs [1], [9], [11], [12] have been proposed. In [9], authors use a parallel topological sorting algorithm to set the levels and use a counter-based scheduling mechanism, where each element only waits for its own dependencies. In [11], authors propose a simple preprocessing phase, where self-scheduling mechanism is set up based on the in-degree of dependency graph nodes.

The drawbacks of level-set and self-scheduling approaches are as follows:

- A level consists of rows that can be run in parallel and a synchronization barrier at the end of each level is required, causing the levels to be computed serially in a sequence. Therefore, using barriers incurs high synchronization overhead, especially for matrices with large number of levels. Computation of rows in a successor level has to wait for the current level to reach the barrier even if all dependencies of a row in a successor level are satisfied.
- Self-scheduling algorithms partition the rows into fine-grained tasks and a task is launched when its input data is ready. While this method eliminates the need for synchronization barriers between consecutive levels, its implementation typically requires threads to busy-wait on their predecessor.

In [16], both level set and synchronization-free methods are utilized in a hybrid approach is developed for CPUs. In our previous study [24], we adopted a similar approach to [16]. Both works aim to reduce synchronization points for SpTRSV on CPUs, eliminating unnecessary dependencies.

Another notable body of work about optimizing SpTRSV is done in compiler domain. In [17], authors enable context-driven optimizations such as matrix reordering by examining the properties of the sparse matrix and generate code specialized for the matrix. Sympiler [4], analyzes the sparse codes symbolically during the compilation process and generates domain-specific code for various sparse matrix operations. In addition to the domain-specific optimizations similar to Sympiler, the specialized code generator proposed in this paper aims to reduce memory accesses by embedding the memory accesses into the code as constants.

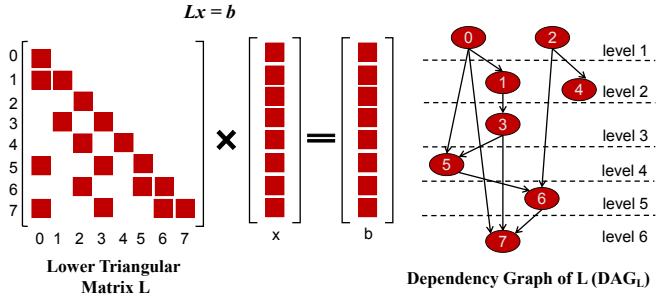
Graph coloring is an NP-Complete problem that requires additional pre-processing and it is used in the optimization of SpTRSV. Authors proposed algebraic block multicolor ordering in [7] for CPUs. Authors of [21] and [15] apply graph coloring to SpTRSV on GPUs. Block-diagonal based methods form another approach to improve SpTRSV. It is a method practiced on CPUs more than compared to GPUs. Matrix is reordered to help forming blocks around the diagonal and off-diagonal to improve the locality [13], [20], [22]. In [22], authors use dense BLAS operations to compute dense off-diagonal blocks. Dense matrix computations have higher parallelism than their sparse counterparts, therefore, creating dense blocks in sparse matrices is an effective method to improve performance. These techniques are highly dependent on the sparsity structure of the matrix.

III. The Equation Rewriting Method

In this work, we take the level set approach and transform the dependency graph of the given matrix by shifting rows from lower to upper levels using the equation rewriting method. However, the method is also applicable to synchronization-free approach since it works directly on the dependency graph. Transformation can take place to apply load balancing on the tasks too.

The dependency graph is transformed by rewriting the equations each row represents leading to the fattening of the thin levels (levels with a few rows) and removing the thin levels completely, reducing the number of the synchronization barriers. In return, the idle cores can be utilized in the fattened levels increasing the parallelism. Rewritten rows are calculated which normally must be calculated way after (due to dependencies) at the cost of increased floating point operations (FLOPS) and sometimes increased memory access.

Figure 2 demonstrates the graph transformation process. In this example, equation rewriting has been applied to row 3 ($x[3]$) twice. The original equations of row 0 and row 1 are colored in green and blue. The first rewriting operation is also blue since it uses row 1, and the second rewriting operation is green since it uses row 0. The dependency graph on the left is the original dependency graph, while the ones in the middle and on the right show the rows after the equation rewriting operations.



Algorithm 1: Solving SpTRSV serially for $Lz = b$ where L is in CSR format

```
for i = 0 to n-1 do
    for j=rowPtrL[i] to rowPtrL[i+1]-1 do
        sum[colIdxL[j]] += valL[j] * z[colIdxL[j]];
    end
    z[i] = (b[i]-sum[i])/valsL[rowPtrL[i+1]-1];
end
```

Fig. 1. **Left:** Lower triangular matrix L and L's dependency graph partitioned into levels **Right:** Serial algorithm for SpTRSV where the sparse matrix is in CSR format.

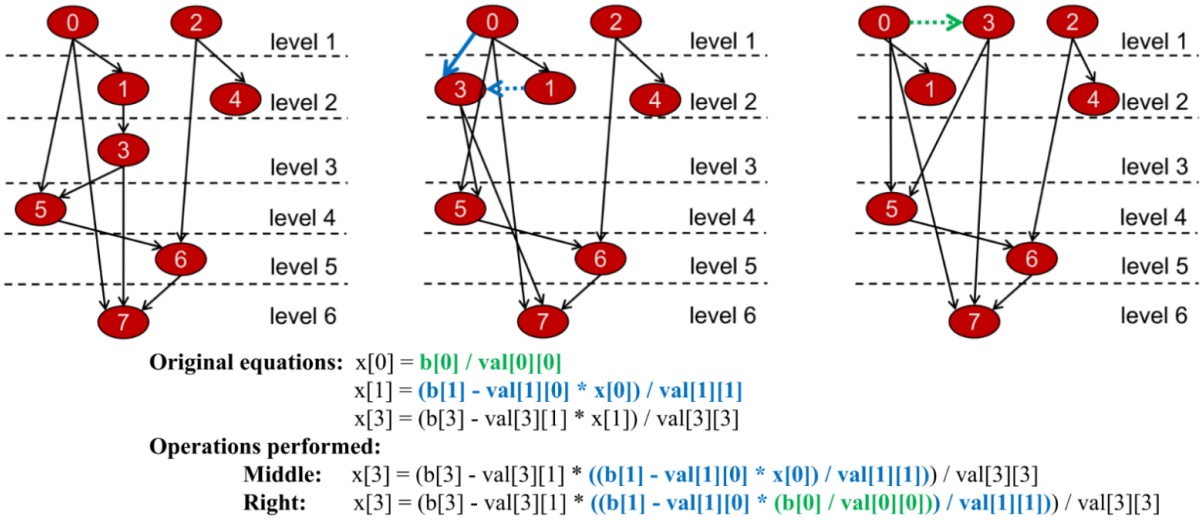


Fig. 2. **Left:** original equations before equation rewriting is applied **Middle:** rewriting row 3 using the original equation of row 1 **Right:** rewriting row 3 further, using the original equation of row 0. The colored and dashed arrows indicate the broken bonds and the colored but steady arrows indicate the newly formed bonds.

In the original dependency graph, row 3 has a dependency on row 1 which depends on row 0. As shown in the middle, by replacing row 1's with its equation within the calculation of row 3, its dependency on row 1 is broken and now it depends on row 0, and row 3 is shifted up from level 3 to level 2. Now, level 3 is empty. Repeating the operation as shown on the right, row 3 is shifted up to level 1 and now does not depend on any row. Since level 3 is now empty it can be removed.

An important point to be noted is that the new equation specified with the label Figure 2 **Right** is not in the $Lx = b$ format anymore. This results in a significant increase in the number of arithmetic operations, hence in FLOPS. More importantly, the equations cannot be represented with a lower triangular matrix anymore. This may lead to losing the opportunity to use existing SpTRSV algorithm for this matrix. To remedy this problem, a reorganization of the rewritten equations will be added to the rewriting process in the future. The equation specified in Figure 2, **Right** is rearranged to be in $Lx = b$ format and it is shown in Figure 3. As seen in this figure, values of b vector are updated and the number of FLOPs decreases.

IV. Matrix Analysis and Code Generation

Matrix analysis module analyzes the input matrix and extracts matrix properties such as number of rows, number of nonzeros. When the analysis of the matrix is complete, DAG of the matrix and the level sets are constructed. Then additional information such as total number of memory accesses per level and average number of memory accesses per level is extracted from the DAG. This information is fed to the code generation module where equation rewriting process kicks in and code specialized for the matrix is generated.

The specialized code generator is in very early stages of development. It produces code that will run in parallel by rewriting the equations of chosen rows and thus transforms the DAG accordingly. The code generator optimizes the code by reducing the memory accesses by converting them to constants and burying them into the code, and by eliminating the indirect indexing for rewritten rows. Several critical optimizations such as load balancing of level workloads, arithmetic optimizations and forming dense blocks to improve the locality are not implemented yet. An example to an arithmetic optimization is when the same memory accesses to vector x appear multiple

x[3] = (b[3] - L[3][1] * ((b[1] - L[1][0] * (b[0] / L[0][0])) / L[1][1])) / L[3][3] (c)

A= b[0] / L[0][0] (1)
x[3] = (b[3] - L[3][1] * ((b[1] - L[1][0] * A) / L[1][1])) / L[3][3] (c')
x[3] = (b[3] - L[3][1] * (b[1]/ L[1][1] - L[1][0] * A / L[1][1]) / L[3][3] (c")
B = b[1]/ L[1][1] (2)
C =L[1][0] * A / L[1][1] (3)
x[3] = (b[3] - L[3][1] * (B - C)) / L[3][3] (c"')
D = L[3][1] * (B - C) (4)
x[3] = (b[3] - D) / L[3][3] (c"")
b[3]' = b[3] - D (5)
x[3] = b[3]' / L[3][3] (c""')

Fig. 3. Rearrangement of the rewritten equation from Figure 2, **Right**. Now the equation is in $Lx = b$ format. The b vector entries are updated. Equations 1-5 are there for following the rearrangement easily.

times in the equation as a result of rewriting operation. These can be grouped together since each of them are only multiplied by different values. In addition, currently the rows to be rewritten are chosen manually by the programmer. The specialized code generator is written in C ++ and the code produced is C code with OpenMP [3] pragmas.

Examples of the code generated are provided below in Figure 4 for matrix lung2 from SuiteSparse Matrix Collection [5]. On the left, the code generated when no equation rewriting is applied is shown. A function per level is generated by the specialized code generator until a threshold is hit: if the level is thick (having many rows), multiple functions are generated. Since equation rewriting is not applied, no rows are shifted between levels and thus no synchronization points are removed. Furthermore, memory accesses to vector x remain and only the others (to matrix L and vector b) are converted to constants. On the right handside, the code generated when equation rewriting is applied is shown. Rows $2, 3, 4$ and 5 which are at levels 1 and 2 are shifted to level 0. Hence, the rows at higher levels depending on the rows $2, 3, 4$ and 5 can be calculated immediately and the levels 1 and 2 are removed together with their synchronization barriers at the end of these levels.

Figure 5 shows the driver code generated by the specialized code generator. The function(s) generated for each level are called by openMP pragma directives for the parallel execution of the generated code. The pragma directives are not generated for a serial execution. All code is generated automatically along with a necessary spTRSV code and a makefile. Hence the executable is built with a single make command.

In both Figure IV and Figure 5 very short or long functions are generated due to the lack of an efficient loaf balancing algorithm.

V. Experiment Results

To observe the affects of the equation rewriting method on the dependency graph, we conducted two experiments. The matrix chosen for this experiment is lung2 from SuiteSparse Matrix Collection [5]. lung2 has $109,460$ rows and $492,564$ nonzeros. After level sets are constructed, lung2 has 478 levels (synchronization barriers) and 94% of these levels have only 2 rows, hence they are very serial. The experiments are conducted on a dual socket Xeon Westmere with 2.53GHz clock speed, 12 cores (24 threads) and 96GB DDR + 16 MCDRAM RAM. As the compiler, clang 5.0 [8] is used.

As mentioned in Section IV, code generator is a prototype providing only reduction in memory accesses and elimination of indirect indexing and important optimizations such as load balancing of level workloads are not implemented yet.

In the first experiment, the performance of the specialized code generator is measured. No equation rewriting is applied and the code generated is serial. Average of 10 iterations was measured as $1.98ms$. As a comparison, a serial handwritten code using again the level set approach has an average of 10 iterations as $1.14ms$. An important note is that since lung2 is a very serial matrix, a serial implementation gives the better performance than its parallel counterparts. We observed that the performance of the prototype falls behind the compared serial code for the following reasons: **1)** the generated code is too long, **2)** there is no organization of the equations, same calculations are repeated, **3)** the code generator is written with generating parallel code in mind and there is no effective load balancing algorithm. Hence, there are unnecessary function calls. Functions can be very short and they should be merged. The aim of this experiment is to create a code skeleton to observe the benefits of the optimizations implemented as well as the optimizations planned.

In the second experiment the merits of the equation rewriting method is observed. As seen in Figure V, we applied equation rewriting to the thin levels (levels with very few rows) removing most of them completely by shifting their rows to upper levels. After rewriting, 478 levels dropped down to 66, removing 86% of the synchronization barriers. The number of total FLOPs to be performed increased only by 10%. In this experiment which is carried out by running the generated code serially, the average of 10 iterations was measured as $2.06ms$.

Considering that most of the proposed optimizations are not implemented, the lack of an effective load balancing method

```
// level 0
void calculate0(double* x) {
 x[0] = 1 / 1;
 x[1] = 85.7849 / 85.7849;
 }

// level 1
void calculate1(double* x) {
 x[2] = (0.0579356-((-2.93613e-07) *
  ↪  x[0]+(0.0579358) * x[1]))/9.6701e-08;
 x[3] = (-163.137-(-248.922) * x[1])/85.7849;
}

// level 2
void calculate2(double* x) {
 x[4] = (0.0579356-((-2.93613e-07) *
  ↪  x[2]+(0.0579358) * x[3]))/9.6701e-08;
 x[5] = (-163.137-(-248.922) * x[3])/85.7849;
}
...
```

```
// level 0
void calculate0(double* x) {
  x[0] = 1 / 1;
  x[1] = 85.7849 / 85.7849;
  x[2] = (0.0579356 - (-0.000000*1.000000 /
   ↪  1.000000 + 0.057936*85.784944 /
   ↪  85.784944)) / 9.6701e-08;
  x[3] = (-163.137 - (-248.922133*85.784944 /
   ↪  85.784944)) / 85.7849;
  x[4] = (0.0579356 - (-0.000000*((0.057936 -
   ↪  (-0.000000*(1.000000 / 1.000000) +
   ↪  0.057936*(85.784944 / 85.784944))) /
   ↪  0.000000) + 0.057936*((-163.137189 -
   ↪  (-248.922133*(85.784944 / 85.784944))) /
   ↪  85.784944))) / 9.6701e-08;
  x[5] = (-163.137 -
   ↪  (-248.922133*((-163.137189 -
   ↪  (-248.922133*(85.784944 / 85.784944))) /
   ↪  85.784944))) / 85.7849;
  ...
}
```

Fig. 4. **Left:** Functions generated for lung2, when no rewriting is applied. Each function corresponds to a level in the original DAG **Right:** Function generated for lung2 for the rows of level 0, when equation rewriting is applied. As a result of the equation rewriting process, rows of levels 1 and 2 are shifted to level 0.

```
#include "calculators0.h"
void run0(double* x) {
#pragma omp parallel num_threads(12){
  #pragma omp sections {
  #pragma omp section
   { calculate0(x) }
   ....
   }
  }
}
```

Fig. 5. Generated code that runs in parallel and calls the function shown on Figure IV, **Left**.

and equations are not reorganized, this experiment shows that **1)** the rewriting operation is promising in terms of removing the synchronization barriers, **2)** matrices which deliver poor performance on parallel architectures and accelerators due to the parts lacking parallelism, can now deliver better performance with fewer and fatter levels, **3)** benefit from the decrease in the number of synchronization barriers and utilizing more cores is likely to surpass the performance loss due to the increase in FLOPs, **4)** starting with an effective load balancing algorithm, optimizations mentioned in Section IV are critical for performance gains.

VI. Conclusion

In this study, a new and original method of rewriting the equations of the rows of is proposed for the optimization of SpTRSV. We aim to increase the parallelism of the parts of a lower triangular matrix lacking parallelism by transforming them with the equation rewriting method. Thus, the dependency graph transformation allows the use of more cores by relaxing or completely eliminating row dependencies. In addition, thin levels can be completely eliminated by this method reducing the need for synchronization barriers. The equation rewriting method proposed is reinforced with a specialized code generator that can enable domain-specific optimization by generating code specialized to the matrix. Performance results show that the rewriting method is promising for performance optimization of SpTRSV.

References

[1] Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. Accelerating the task/data-parallel version of ilupack's bicg in multi-cpu/gpu configurations. *Parallel Computing 85* (2019), 79 – 87.

[2] Anderson, E., and Saad, Y. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing 1*, 1 (1989), 73–95.

[3] Board, T. O. A. R. OpenMP application program interface, November 2015.

[4] Cheshmi, K., Kamil, S., Strout, M. M., and Dehnavi, M. M. Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2017), SC '17, ACM, pp. 13:1–13:13.

[5] Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 1:1–1:25.

[6] Hammond, S. W., and Schreiber, R. Efficient iccg on a shared memory multiprocessor. *International Journal of High Speed Computing 04*, 01 (1992), 1–21.

[7] Iwashita, T., Nakashima, H., and Takahashi, Y. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (May 2012), pp. 474–483.

[8] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis& transformation. CGO '04, IEEE Computer Society, p. 75.

[9] Li, R. ON PARALLEL SOLUTION OF SPARSE TRIANGULAR LINEAR SYSTEMS IN CUDA. Tech. rep., 2017.

[10] Li, R., and Saad, Y. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing 63*, 2 (feb 2013), 443–466.

[11] Liu, W., Li, A., Hogg, J., Duff, I. S., and Vinter, B. A synchronization-free algorithm for parallel sparse triangular solves. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 617–630.

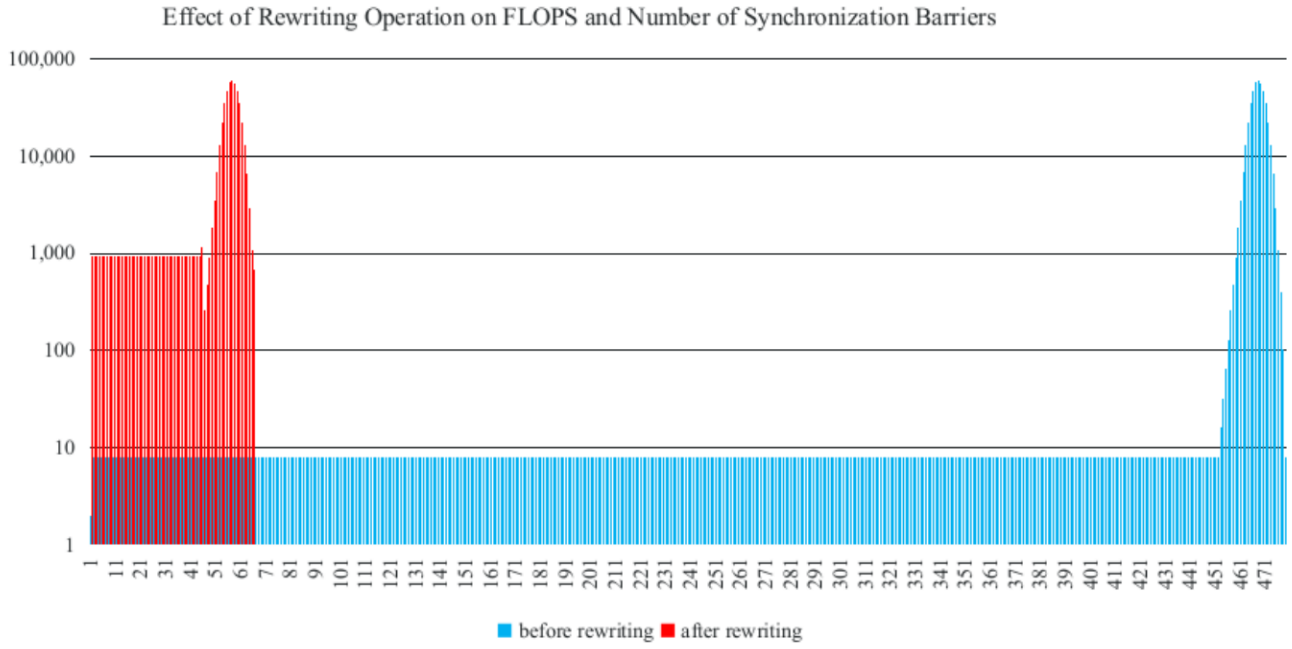


Fig. 6. Number of FLOPS and number of levels before and after the equation rewriting method is applied. Rewriting rows reduces the number of synchronization barriers and increases the number of rows to be computed in a level

[12] LIU, W., LI, A., HOGG, J. D., DUFF, I. S., AND VINTER, B. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience 29*, 21 (2017), e4244. e4244 cpe.4244.

[13] MAYER, J. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing 86*, 4 (Sep 2009), 291.

[14] NAUMOV, M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. Tech. rep., 2011.

[15] NAUMOV, M., CASTONGUAY, P., AND COHEN, J. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. Tech. rep., 2015.

[16] PARK, J., SMELYANSKIY, M., SUNDARAM, N., AND DUBEY, P. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488* (New York, NY, USA, 2014), ISC 2014, Springer-Verlag New York, Inc., pp. 124–140.

[17] RONG, H., PARK, J., XIANG, L., ANDERSON, T. A., AND SMELYANSKIY, M. Sparso: Context-driven optimizations of sparse linear algebra. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)* (Sep. 2016), pp. 247–259.

[18] ROTHBERG, E., AND GUPTA, A. Parallel iccg on a hierarchical memory multiprocessor — addressing the triangular solve bottleneck. *Parallel Computing 18*, 7 (1992), 719 – 741.

[19] SALTZ, J. H. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. Stat. Comput. 11*, 1 (Jan. 1990), 123–144.

[20] SMITH, B., AND ZHANG, H. Sparse triangular solves for ilu revisited: Data layout crucial to better performance. *Int. J. High Perform. Comput. Appl. 25*, 4 (Nov. 2011), 386–391.

[21] SUCHOSKI, B., SEVERN, C., SHANTHARAM, M., AND RAGHAVAN, P. Adapting sparse triangular solution to gpus. In *2012 41st International Conference on Parallel Processing Workshops* (Sep. 2012), pp. 140–148.

[22] TOTONI, E., HEATH, M. T., AND KALE, L. V. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing 40*, 9 (2014), 454 – 470.

[23] WANG, X., LIU, W., XUE, W., AND WU, L. swsptrsv: A fast sparse triangular solve with sparse level tile layout on sunway architectures. *SIGPLAN Not. 53*, 1 (Feb. 2018), 338–353.

[24] YILMAZ, B., SIPAHIOĞLU, B., AHMAD, N., AND UNAT, D. Adaptive level binning: A new algorithm for solving sparse triangular systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (New York, NY, USA, 2020), HPCAsia2020, Association for Computing Machinery, p. 188–198.