A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves

Weifeng Liu[1,2], Ang Li[3], Jonathan Hogg[2], Iain S. Duff[2], and Brian Vinter[1]

[1] Niels Bohr Institute, University of Copenhagen, Denmark
[2] Scientific Computing Department, STFC Rutherford Appleton Laboratory, UK
[3] Eindhoven University of Technology, Netherlands

Abstract. The sparse triangular solve kernel, SpTRSV, is an important building block for a number of numerical linear algebra routines. Parallelizing SpTRSV on today's manycore platforms, such as GPUs, is not an easy task since computing a component of the solution may depend on previously computed components, enforcing a degree of sequential processing. As a consequence, most existing work introduces a preprocessing stage to partition the components into a group of level-sets or colour-sets so that components within a set are independent and can be processed simultaneously during the subsequent solution stage. However, this class of methods requires a long preprocessing time as well as significant runtime synchronization overhead between the sets. To address this, we propose in this paper a novel approach for SpTRSV in which the ordering between components is naturally enforced within the solution stage. In this way, the cost for preprocessing can be greatly reduced, and the synchronizations between sets are completely eliminated. A comparison with the state-of-the-art library supplied by the GPU vendor, using 11 sparse matrices on the latest GPU device, show that our approach obtains an average speedup of 2.3 times in single precision and 2.14 times in double precision. The maximum speedups are 5.95 and 3.65, respectively. In addition, our method is an order of magnitude faster for the preprocessing stage than existing methods.

1 Introduction

The sparse triangular solve kernel, SpTRSV, is an important building block in a number of numerical linear algebra routines, such as direct methods [5, 7], preconditioned iterative methods [22], and least squares problems [3]. This operation computes a dense solution vector x from a sparse linear system $Lx = b$, where L is a square lower triangular sparse matrix and b is a dense vector.

Compared to a dense triangular solve [9] and other sparse basic linear algebra subprograms (BLAS) [8, 14] such as sparse transposition [27], sparse matrix-vector multiplication [16, 17] and sparse matrix-matrix multiplication [15], the SpTRSV operation is more difficult to parallelize since it is inherently sequential. This means that, for a lower triangular sparse matrix, computing any single component x_k may depend on having first computed a subset of previous components

$x_0, \cdots, x_{k-1}$. Therefore, most existing research concentrates on adding a preprocessing stage to divide the entries of x into a number of sets (known as level-sets or colour-sets). Even though the sets have to be executed in sequence, entries in any single set can be computed in parallel. As a result, parallel hardware can be exploited efficiently. This class of methods demonstrates great advantage over the original sequential implementation both on CPUs [10, 20, 24, 28] and on GPUs [12, 19, 26].

However, the set-based methods have two performance bottlenecks. Firstly, finding a good set partitioning often takes too much time, which may offset or even wipe out the benefits from parallelization. Secondly, the synchronization between consecutive sets reduces parallelization efficiency at runtime. In fact, due to these large overheads, finding an efficient thread synchronization scheme still remains a popular research topic for computer design [11, 13, 21].

In this paper, we propose a synchronization-free algorithm for parallel SpTRSV on GPUs. Our approach requires only a light-weight preprocessing stage without set partitioning. More importantly, our method completely eliminates the runtime barrier synchronizations among sets. By doing so, our method resolves the bottlenecks and achieves significant performance improvement. Using 11 sparse matrices from the University of Florida Sparse Matrix Collection [6], our method achieves an average speedup of 2.3 times in single precision and 2.14 times in double precision over the vendor provided library on the latest NVIDIA GPU. The maximum speedups are 5.95 and 3.65, respectively. More noticeably, the preprocessing stage of our algorithm is on average 43.7 faster (maximum of 70.5 times) than existing set-based methods in the vendor supplied library.

2 Background

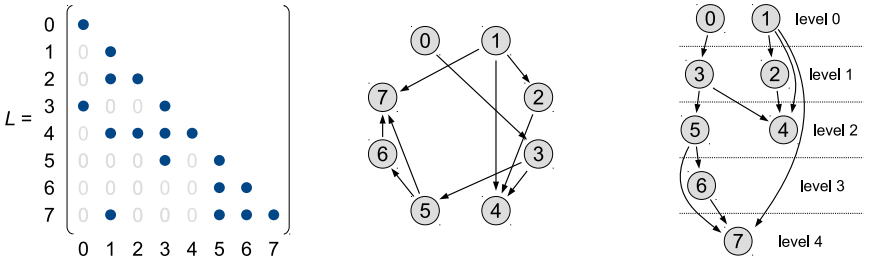
2.1 Serial Algorithm

Without loss of generality, in the paper we assume that the input matrix L is a nonsingular lower triangular matrix and is stored in the *compressed sparse column* (CSC) format composed of three arrays `col_ptr`, `row_idx` and `val`. A typical serial implementation of SpTRSV for solving $Lx = b$ is given in Algorithm 1. This method traverses all columns in ascending order (line 3) and solves a single component of x in each step (line 4). After that, the code updates all the positions corresponding to the nonzero entries of the current column in an intermediate array `left_sum` (lines 5–7).

As can be seen, the columns in the main *for* loop (lines 3–8) cannot be parallelized as the ith column requires the ith value in `left_sum` (line 4), which may be affected by previous columns that update `left_sum[`i`]` (line 6). To be clear, we give an example. Figure 1 (a) shows a matrix L, for which the underlying dependencies are illustrated in graph form in Figure 1 (b). Obviously, vertex 5 (i.e., x_5) cannot be solved before vertex 3 is solved, and vertex 3 has to wait for vertex 0.

Algorithm 1 A serial SpTRSV method for $Lx = b$, where L in CSC format.

```
1: MALLOC(*left_sum, n)
2: MEMSET(*left_sum, 0)
3: for i = 0 to n − 1 do
4:     x[i] ← (b[i]-left_sum[i])/val[col_ptr[i]]
5:     for j = col_ptr[i]+1 to col_ptr[i + 1]−1 do
6:         left_sum[row_idx[j]] ← left_sum[row_idx[j]] + val[j] × x[i]
7:     end for
8: end for
9: FREE(*left_sum)
```



(a) L's matrix form. (b) L's graph form. (c) Level-sets generated.

Fig. 1. A lower triangular matrix L and parallel SpTRSV using the level-set method.

2.2 Level-Set Method for Parallel SpTRSV

The motivation of parallel-SpTRSV comes from the observation that some components/vertices are independent and can be processed simultaneously (e.g., vertices 0 and 1 in Figure 1 (b)). Therefore, the components can be partitioned into a number of sets so that components inside a set can be solved in parallel, while the sets are processed sequentially (i.e., level by level). With this observation, Anderson and Saad [1] and Saltz [23] introduced a preprocessing stage to perform such a partition before the solving stage. Figure 1 (c) shows that five level-sets are generated for the matrix L. Consequently, levels 0, 1 and 2 can use parallel hardware (e.g., a dual-core machine) for accelerating SpTRSV. However, between sets, dependencies still exist so synchronization is required at runtime.

2.3 Motivation for Avoiding Synchronization

Synchronization remains a performance bottleneck for many applications and has long been a classic problem in computer systems research [11, 13, 21]. To evaluate the synchronization cost in SpTRSV, we run a parallel SpTRSV implemented by Park et al. [20] based on the aforementioned level-set approach. We show the cost of the preprocessing stage and a breakdown of the solving stage execution time (i.e., synchronization cost and floating-point calculations)

using four representative matrices[4] from the University of Florida Sparse Matrix Collection [6].

Matrix name	Preprocessing cost (ms)	SpTRSV cost (ms)	SpTRSV cost breakdown (ms)		#Level-sets
			Synchronization	Compute	
FEM/ship_003	92.46	12.95	10.96	1.99	4367
FEM/Cantilever	47.89	9.60	5.62	3.98	2397
chipcool0	8.74	1.99	1.15	0.84	534
nlpkkt160	484.67	38.30	0.01	38.29	2

Table 1. Breakdown of naïve level-set method [20] on Intel dual-socket E5-2695 v3.

We have two observations from Table 1. Firstly, the preprocessing stage takes much longer than a single call to SpTRSV. Specifically, the preprocessing stage is 4.39 times (matrix *chipcool0*) to 12.65 times (matrix *nlpkkt160*) slower than the main kernel of SpTRSV. This implies that if SpTRSV is only executed a few times, level-set based parallelization is not attractive. Secondly, when the number of level-sets increases, the overhead for synchronization dominates the SpTRSV solving stage execution time. For example, matrix *FEM/ship_003* has 4367 level-sets that implies 4366 explicit barrier synchronizations in the solving stage and accounts for 85% of the total SpTRSV execution time (10.96 ms out of 12.95 ms). In contrast, the synchronization overhead for matrix *nlpkkt160* is much less as only two level-sets are generated.

Therefore, to improve the performance of parallel SpTRSV, it is crucial to reduce the overhead for preprocessing (i.e., generating level-sets) and to avoid the runtime barrier synchronizations.

3 Synchronization-Free Algorithm

The objective of this work is to eliminate the cost for generating level-sets and the barrier synchronizations between the sets. Due to the inherent dependencies among components, the major task for parallelizing SpTRSV is to clarify such dependencies and to be sure to respect them when solving at runtime.

In this work, we use GPUs as the platform to exploit inherent parallelism when there are many components for a very large matrix. We assign a warp of threads to solve a single component of x (a *warp* is a unit of 32 SIMD threads executed in lock-step for NVIDIA GPUs. For AMD GPUs the warp is 64 threads and is denoted by the term *wavefront*). To respect the partial order of SpTRSV, we need to be sure that the warps associated with dependent entries (if any) must be finished first. Thus thread-blocks of multiple warps are required to be dispatched in ascending order, even though they can be switched and finished in arbitrary order. Since the partial order is essentially **unidirectional** (i.e., any component only depends on previous components but not on later ones, see Figure 1 (b)), we can map entries to warps and strictly respect the partial order of the entries so that no warp execution deadlock will occur.

[4] Similar to [20], the nonsingular matrix L is the lower triangular part of the input matrix, plus a dense main diagonal.

Therefore, before actually solving for a particular component, we let the processing warp learn how many entries have to be computed in advance (i.e., the number of dependent entries). This number equals the in-degree of a vertex in the graph representation of a matrix (Figure 1 (b)), which is also identical to the number of nonzero entries of the current matrix row minus one (to exclude the entry on diagonal). Thus, we use an intermediate array `in_degree` of size n to hold the number of nonzero entries for each row of the matrix. This is all we do in the preprocessing stage. Algorithmically, this step is part of transposing a sparse matrix in parallel [27]. Compared with the complex dependency extraction in the set-based methods that have to analyse the sparsity structure, our method requires much less work. Lines 3–7 in Algorithm 2 show the pseudocode of our preprocessing stage.

Algorithm 2 The proposed synchronization-free SpTRSV algorithm.

```
1: MALLOC(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, n)
2: MEMSET(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, 0)
3: function PREPROCESSING-STAGE()
4:     for i = 0 to nnz − 1 in parallel do
5:         ATOMIC-ADD(&d_in_degree[row_idx[i]], 1)
6:     end for
7: end function
8: function SOLVING-STAGE()
9:     th ← SET()                                          ▷ size of diagonal block
10:    for i = 0 to n − 1 in parallel do          ▷ One concurrent warp for one component.
11:        while s_in_degree[i]+1 ≠ d_in_degree[i] do
12:            //busy wait
13:        end while
14:        x[i] ← (b[i]-d_left_sum[i]-s_left_sum[i])/val[col_ptr[i]]
15:        for j = col_ptr[i]+1 to col_ptr[i + 1]−1 in parallel do ▷ One thread for one nonzero.
16:            rid ← row_idx[j]
17:            if rid < i + th − i%th then      ▷ Use on-chip scratchpad for red areas in Figure 3.
18:                ATOMIC-ADD(&s_left_sum[rid], val[j] × x[i])
19:                ATOMIC-ADD(&s_in_degree[rid], 1)
20:            else                             ▷ Use off-chip memory for green area in Figure 3.
21:                ATOMIC-ADD(&d_left_sum[rid], val[j] × x[i])
22:                ATOMIC-SUB(&d_in_degree[rid], 1)
23:            end if
24:        end for
25:    end for
26: end function
27: FREE(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree)
```

Knowing the in-degree information indicating how many warps have to be finished in advance, we can initiate sufficient numbers of warps to fully exploit the irregular parallelism. For an arbitrary warp, after finishing the necessary floating-point computation (line 14 in Algorithm 2) for a component, it notifies all the later entries that depend on the current one, by atomic updating (lines 19 and 22). Note that atomic operations are needed here as multiple updates from different warps may happen simultaneously. Therefore, a warp only has to wait (lines 11–13) until its corresponding in-degrees are all eliminated, implying that all the dependent components are successfully solved and the warp can start processing safely. Due to the warp multi-issuing property of GPUs, a warp can start processing immediately after its dependencies have been satisfied, without

any false waiting incurred by the hardware. Also, the first component of x can be solved without any dependencies.

Figure 2 illustrates the procedure of our synchronization-free algorithm[5] using an example. Suppose there are three warps enrolled, tagged as warp0, warp1 and warp2. They follow the same procedure and are context-switched by the hardware scheduler. For an arbitrary warp, the central region contained in the red dotted box (labelled as the critical section protecting the `left_sum` array) separates the whole procedure into three phases: *lock-wait, critical section* and *lock-update*.

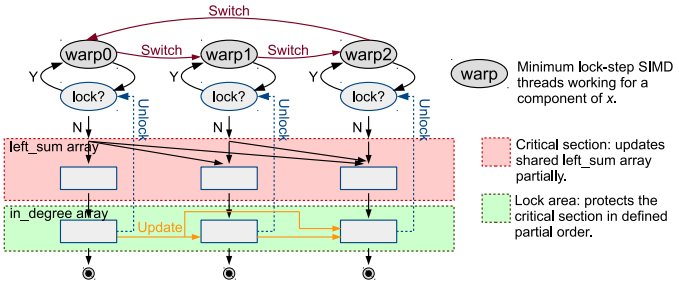


Fig. 2. The basic procedure of our synchronization-free algorithm.

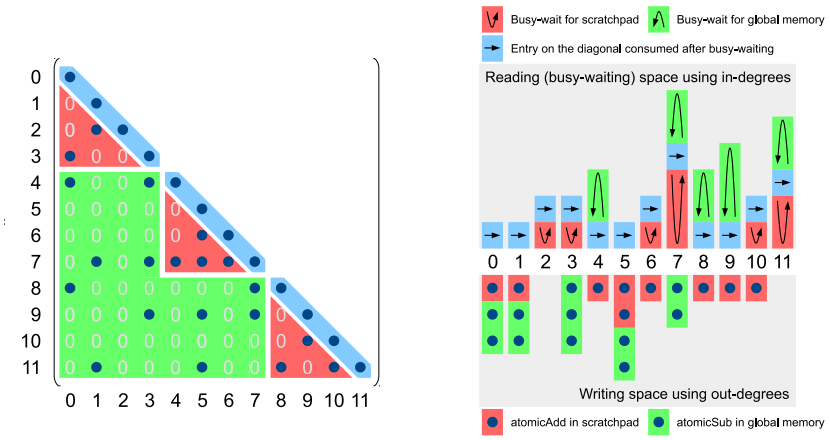
In the **lock-wait** phase, the warp iteratively evaluates the status of the lock protecting the critical section of the current warp. If locked, it waits in the loop (known as *spinning*); otherwise, it stops waiting and enters the next phase. Although the lock here is a spin-lock, it does not have the busy-waiting problem. Based on our observation, if the *clock()* function is invoked inside the waiting loop, the hardware warp scheduler will be signalled to switch to the next warp context. This avoids the execution deadlock. In the **critical section** phase, the warp updates the components in `left_sum` that have dependencies on the components the warp is currently working on. This is done in an order that depends on the partial dependency defined by the sparsity structure. After that, it aborts the critical section and enters the lock-update phase. In the last **lock-update** phase, the warp updates the dependent `in_degree` array, in the same order as for the `left_sum` (so that all the order dependencies are strictly respected). The warp updates the related in-degrees. Depending on the number of components in that column (line 15 in Algorithm 2), it may require one or several updates. When an in-degree is updated to reach the target value (so that all the dependencies of the component are resolved), the lock corresponding to that in-degree is unlocked. Consequently, the warp waiting for that lock can abort the waiting phase and enter its critical section.

Lines 8–26 in Algorithm 2 give the pseudocode for the solving stage of our synchronization-free SpTRSV method. An optimization here is to exploit the

[5] Note that hardware-level synchronizations in atomic operations should not be confused with barrier synchronizations in the set-based methods, when we claim that the proposed method is synchronization-free.

GPU on-chip scratchpad memory. The idea is to allocate two sets of intermediate arrays, one on local scratchpad memory (`s_left_sum` and `s_in_degree`) and the other on off-chip global memory (`d_left_sum` and `d_in_degree`), see line 1 of Algorithm 2. When a warp finds a dependent entry (the later entry that depends on the current one) is in the same GPU thread-block composed of multiple warps, it updates the local arrays (lines 18–19) in the scratchpad memory for faster accessing. Otherwise, it updates the remote off-chip arrays (lines 21-22), to notify warps from other thread-blocks. The sum of the two arrays (line 11) is used to verify if all the dependencies are fulfilled ultimately.

Figure 3 (a) shows an example using 12 warps organized in 3 thread-blocks for solving a system of order 12×12. Operations in on-chip scratchpad memory are marked red (lines 18–19 in Algorithm 2), other operations in off-chip memory are marked green (lines 21–22), and the diagonal entries are coloured blue (line 14). Figure 3 (b) plots read/write behaviours for solving the 12 components (presented as 12 columns) of x. We can see that entries 0, 1 and 5 can be solved immediately once the corresponding warps are issued since they have no in-degree (see the top half of the subfigure), and they update values using their out-degrees (see the bottom half). In contrast, the other entries have to busy-wait until their in-degrees are eliminated.



(a) Matrix. (b) Read/write behaviours.

Fig. 3. An example of the proposed synchronization-free SpTRSV method. The red area performs atomic-adds (lines 18–19 in Algorithm 2) in scratchpad memory, and the green area performs both atomic-adds (line 21) and atomic-subs (line 22) in off-chip memory.

4 Experimental Results

4.1 Experimental Setup

We have implemented the proposed synchronization-free SpTRSV method both in CUDA and in OpenCL, and have evaluated it on three GPUs: (1) an NVIDIA Tesla K40c GPU of Kepler architecture, (2) an NVIDIA GeForce GTX Titan X GPU of newer Maxwell architecture, and (3) an AMD Radeon R9 Fury X GPU of GCN architecture. As references, we also benchmark the most recent SpTRSV implementations from two libraries cuSPARSE v7.5 and MKL v11.3 Update 1 provided by NVIDIA and Intel, respectively.

Because mixed-precision numerical methods have recently attracted much attention, we evaluate all methods in both single and double precision. Information about the platforms and test schemes are listed in Table 2.

The testbeds	The participating SpTRSV algorithms
A dual-socket Intel Xeon E5-2695 v3 (Haswell, 2×14 cores @ 2.3 GHz, 128 GB ECC DDR4 @ 2×68.3 GB/s).	(1) The `mkl_?csrtrsv` in MKL v11.3 Update 1. Note that this is a highly tuned serial implementation. (2) The parallel executor `mkl_sparse_?_trsv` using the functions `mkl_sparse_set_sv_hint` and `mkl_sparse_optimize` as an inspector in MKL v11.3 Update 1.
An NVIDIA Tesla K40c (Kepler GK110B, 2880 CUDA cores @ 0.75 GHz, 12 GB GDDR5 @ 288 GB/s, driver v352.39).	(1) The latest SpTRSV method `cusparse?csrsv2_solve` using functions `cusparse?csrsv2_bufferSize` and `cusparse?csrsv2_analysis` in its preprocessing stage in the NVIDIA cuSPARSE v7.5. (2) The synchronization-free method proposed in this paper.
An NVIDIA GeForce GTX Titan X (Maxwell GM200, 3072 CUDA cores @ 1 GHz, 12 GB GDDR5 @ 336.5 GB/s, driver v352.39).	(1) The latest SpTRSV method `cusparse?csrsv2_solve` using functions `cusparse?csrsv2_bufferSize` and `cusparse?csrsv2_analysis` in its preprocessing stage in the NVIDIA cuSPARSE v7.5. (2) The synchronization-free method proposed in this paper.
An AMD Radeon R9 Fury X (GCN Fiji, 4096 Radeon cores @ 1.05 GHz, 4 GB HBM @ 512 GB/s, driver v15.12).	(1) The synchronization-free method proposed in this paper.

Table 2. The testbeds and participating SpTRSV algorithms.

Table 3 lists 11 sparse matrices used for our experiments on all platforms. These matrices have also been used in other research on sparse matrix computations [10, 14, 15, 16, 17, 20] and are publicly available from the University of Florida Sparse Matrix Collection [6] (except matrix *Dense*). The selected matrices cover a wide range for the number of level-sets as well as the average parallelism inside a level-set. For example, matrix *nlpkkt160* has only two level-sets so that the computation of most its components can run in parallel, whereas for the matrix *Dense* every component has to wait for earlier components.

4.2 SpTRSV Performance

Figure 4 shows the single and double precision SpTRSV performance on the 11 matrices measured on the four platforms. Overall, the MKL and cuSPARSE libraries show comparable performance, while our synchronization-free method is

Matrix name	#Rows/Columns	#Nonzeros	#Level-sets	Parallelism
nlpkkt160	8,345,600	229,518,112	2	4,172,800
road_central	14,081,816	33,866,826	59	238,675
road_usa	23,947,347	57,708,624	77	311,004
webbase-1M	1,000,005	3,105,536	514	1,946
wiki-Talk	2,394,385	5,021,410	522	4,587
chipcool0	20,082	281,150	534	37
Dense	2,000	4,000,000	2000	1
FEM/Cantilever	62,451	4,007,383	2397	26
crankseg_1	52,804	10,614,210	4056	13
FEM/ship_003	121,728	8,086,034	4367	28
hollywood-2009	1,139,905	113,891,327	82,735	14

Table 3. The benchmark suite.

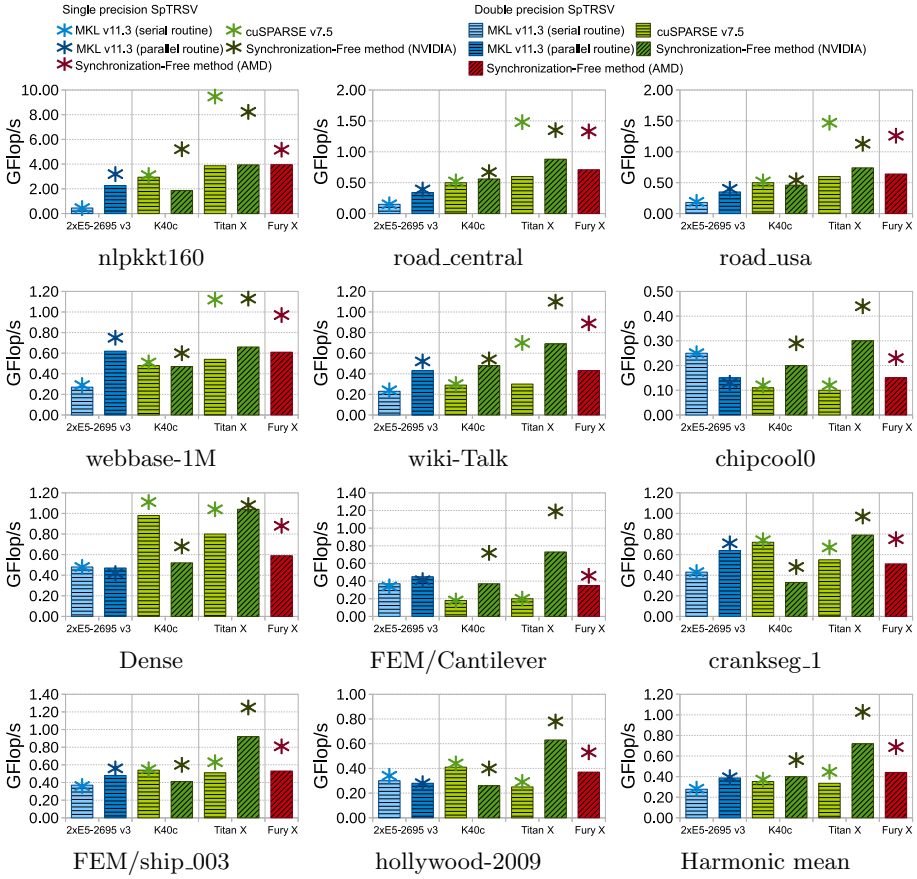


Fig. 4. The SpTRSV performance of the 11 matrices on the four platforms.

much faster (in particular on the Maxwell-based Titan X GPU) than the vendor supplied libraries.

Specifically, on the Titan X GPU, our synchronization-free algorithm demonstrates an average speedup over the cuSPARSE library of 2.3 times in single precision and 2.14 times in double precision. The maximum speedups are 5.95 and 3.65, respectively. The best speedups are from a relatively regular matrix *FEM/Cantilever* that has most of its nonzero entries in its diagonal blocks. For this matrix, the optimizing strategy of using both scratchpad and off-chip mem-

ory improves the overall performance. Also, our method achieves speedups of 2.69 and 2.52 for single and double precision, respectively, for matrix *hollywood-2009*. This matrix requires 82,735 runtime synchronizations (see Table 3) limiting its performance from the level-set methods. In contrast, our method avoids synchronizations and thus obtains much superior performance. For the same reason, our method shows comparable performance compared to existing methods on matrix *nlpkkt160*, which requires only two runtime synchronizations.

Compared to the Kepler based K40c GPU, the Titan X GPU offers higher performance. The major reason is that the Maxwell architecture dramatically improves its micro-architectures for faster atomic operations, which are extensively utilized in our approach. Actually, Scogland and Feng [25] also confirmed that atomic operations have been continuously improved in the last generations of modern GPUs. Moreover, although the AMD Fury X GPU has higher bandwidth than the NVIDIA Titan X, it is in general slower for our synchronization-free SpTRSV algorithm. The main reason may be the difference between the warp/wavefront scheduling strategies on the NVIDIA and AMD GPUs.

4.3 Overhead for Preprocessing

Table 4 shows the preprocessing overhead of the parallel SpTRSV implementations from MKL, cuSPARSE and our approach on the four platforms. As can be seen, our method achieves an average speedup of 43.7 (maximum of 70.5) over the method in cuSPARSE library on the Titan X card. On the K40c device, the speedups are on average 58.2 with a maximum of 89.2. The major reason is that the vendor supplied implementation attempts to find level-sets in the preprocessing phase. Moreover, the AMD Fury X GPU offers lower cost for preprocessing, due to more cores and higher off-chip memory bandwidth.

Matrix name	Intel 2xE5-2695 v3	NVIDIA K40c		NVIDIA Titan X		AMD Fury X
	MKL	cuSPARSE	Sync-Free	cuSPARSE	Sync-Free	Sync-Free
nlpkkt160	64.43	40.58	7.27	19.99	8.91	5.58
road_usa	155.48	160.41	5.06	84.01	3.37	2.31
road_central	92.16	82.01	9.28	42.62	6.98	5.53
wiki-Talk	17.38	16.27	0.33	10.49	0.20	0.16
webbase-1M	7.08	8.53	0.19	5.48	0.13	0.11
chipcool0	1.05	1.48	0.02	1.41	0.02	0.02
FEM/ship_003	9.14	6.41	0.19	4.34	0.26	0.13
FEM/Cantilever	9.52	8.92	0.10	8.28	0.16	0.07
hollywood-2009	223.54	139.98	5.20	204.10	4.82	2.78
crankseg_1	9.30	8.93	0.24	6.14	0.43	0.14
Dense	9.29	3.46	0.08	2.99	0.12	0.05
Harmonic mean	6.80	6.99	0.12	5.71	0.13	0.10

Table 4. Preprocessing cost (in millisecond) of the tested methods on the four platforms.

5 Related Work

Existing parallel SpTRSV methods can be classified into two groups: those constructing level-sets and those generating colour-sets.

Anderson and Saad [1] and Saltz [23] proposed that **level-sets** can expose parallelism in SpTRSV. A few recently developed parallel SpTRSV implementations have improved the level-set method for better data locality and faster synchronization [10, 20, 28]. Maumov [19] implemented the level-set method on NVIDIA GPUs with a tradeoff for decreasing the number of synchronizations. Li and Saad [12] demonstrated that reordering the input matrix can further improve parallelism but requires longer preprocessing time. Unlike the above level-set methods, our synchronization-free SpTRSV algorithm does not analyse the sparsity structure of the input matrix and thus completely removes costs for generating sets and executing barrier synchronization. As a result, our method shows much better performance than level-set methods.

Schreiber and Tang [24] first used graph colouring for constructing **colour-sets** for SpTRSV on multiprocessors. When the input sparse matrix is coloured, it is reorganized as multiple triangular submatrices located on its diagonal. Because all the submatrices can be solved in parallel, this method can be very efficient in practice. Suchoski et al. [26] recently extended the graph colouring method for SpTRSV to GPUs. However, as graph colouring is known to be an NP-complete problem, finding good colour-sets for SpTRSV is in general more time consuming. Thus it may be impractical for real-world applications.

There are also several classes of methods that do not create sets in advance. Mayer [18] pointed out that **2D decomposition** can accelerate SpTRSV but needs to reorganize the data structure of the input matrix. Chow and Patel [4] and Anzt et al. [2] recently developed several **iterative methods** for SpTRSV for use with incomplete factorization. Because iterative methods only give approximate solutions, they should not be used more generally for other scenarios such as using SpTRSV in sparse direct solvers. In contrast, the method we propose in this paper uses the unchanged CSC sparse matrix format and works for general problems.

Some researchers have also utilized atomic operations for **improving fundamental algorithms** such as bitonic sort [29], prefix-sum scan [30], wavefront [11], sparse transposition [27], and sparse matrix-vector multiplication [14, 16, 17]. Unlike those problems, the SpTRSV operation is inherently serial and thus more irregular and complex. We also use atomic operations both in on-chip and off-chip memory, and set atomic operations as the central part of the whole algorithm.

6 Conclusions

In this paper, we have proposed a synchronization-free algorithm for parallel SpTRSV. The method completely eliminates the overhead for generating level-sets or colour-sets (in the preprocessing stage) and for explicit runtime barrier synchronization (in the solving stage). Experimental results show that our approach makes preprocessing an order of magnitude faster than level-set methods, and gives average speedups of 2.3 (with a maximum of 5.95) and 2.14 (with a

maximum of 3.65) over vendor supplied parallel routines for single and double precision SpTRSV, respectively.

Acknowledgments

The authors would like to thank our anonymous reviewers for their invaluable feedback. We also thank Shuai Che for helpful discussion about OpenCL programming, and thank Huamin Ren for supplying access to the machine with the NVIDIA GeForce Titan X GPU. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 671633.

References

[1] Anderson, E., Saad, Y.: Solving Sparse Triangular Linear Systems on Parallel Computers. International Journal of High Speed Computing 1(1), 73–95 (1989)
[2] Anzt, H., Chow, E., Dongarra, J.: Iterative Sparse Triangular Solves for Preconditioning. In: Euro-Par 2015: Parallel Processing. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2015)
[3] Björck, Å.: Numerical Methods for Least Squares Problems. Society for Industrial and Applied Mathematics (1996)
[4] Chow, E., Patel, A.: Fine-Grained Parallel Incomplete LU Factorization. SIAM Journal on Scientific Computing 37(2), C169–C193 (2015)
[5] Davis, T.: Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics (2006)
[6] Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (dec 2011)
[7] Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Inc. (1986)
[8] Duff, I.S., Heroux, M.A., Pozo, R.: An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. ACM Trans. Math. Softw. 28(2), 239–267 (2002)
[9] Hogg, J.D.: A Fast Dense Triangular Solve in CUDA. SIAM Journal on Scientific Computing 35(3) (2013)
[10] Kabir, H., Booth, J.D., Aupy, G., Benoit, A., Robert, Y., Raghavan, P.: STS-k: A Multilevel Sparse Triangular Solution Scheme for NUMA Multicores. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 55:1–55:11. SC '15 (2015)
[11] Li, A., van den Braak, G.J., Corporaal, H., Kumar, A.: Fine-Grained Synchronizations and Dataflow Programming on GPUs. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 109–118. ICS '15 (2015)
[12] Li, R., Saad, Y.: GPU-Accelerated Preconditioned Iterative Linear Solvers. The Journal of Supercomputing 63(2), 443–466 (2013)
[13] Liang, C.K., Prvulovic, M.: MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture. pp. 414–426. ISCA '15 (2015)
[14] Liu, W.: Parallel and Scalable Sparse Basic Linear Algebra Subprograms. Ph.D. thesis, University of Copenhagen (2015)

[15] Liu, W., Vinter, B.: A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. Journal of Parallel and Distributed Computing 85, 47–61 (2015)
[16] Liu, W., Vinter, B.: CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In: Proceedings of the 29th ACM International Conference on Supercomputing. pp. 339–350. ICS '15 (2015)
[17] Liu, W., Vinter, B.: Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. Parallel Computing 49, 179–193 (2015)
[18] Mayer, J.: Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices. Computing 86(4), 291–312 (2009)
[19] Naumov, M.: Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Tech. rep., NVIDIA (2011)
[20] Park, J., Smelyanskiy, M., Sundaram, N., Dubey, P.: Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In: Supercomputing, Lecture Notes in Computer Science, vol. 8488, pp. 124–140. Springer International Publishing (2014)
[21] Ros, A., Kaxiras, S.: Callback: Efficient Synchronization Without Invalidation with a Directory Just for Spin-waiting. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture. pp. 427–438. ISCA '15 (2015)
[22] Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edn. (2003)
[23] Saltz, J.H.: Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. SIAM Journal on Scientific and Statistical Computing 11(1), 123–144 (1990)
[24] Schreiber, R., Tang, W.P.: Vectorizing the Conjugate Gradient Method. In: Proceedings of the Symposium on CYBER 205 Applications (1982)
[25] Scogland, T.R., Feng, W.c.: Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. pp. 63–74. ICPE '15 (2015)
[26] Suchoski, B., Severn, C., Shantharam, M., Raghavan, P.: Adapting Sparse Triangular Solution to GPUs. In: Proceedings of the 2012 41st International Conference on Parallel Processing Workshops. pp. 140–148. ICPPW '12 (2012)
[27] Wang, H., Liu, W., Hou, K., Feng, W.c.: Parallel Transposition of Sparse Data Structures. In: Proceedings of the 30th ACM International Conference on Supercomputing. ICS '16 (2016)
[28] Wolf, M.M., Heroux, M.A., Boman, E.G.: Factors Impacting Performance of Multithreaded Sparse Triangular Solve. In: High Performance Computing for Computational Science – VECPAR 2010, Lecture Notes in Computer Science, vol. 6449, pp. 32–44. Springer Berlin Heidelberg (2011)
[29] Xiao, S., Feng, W.c.: Inter-Block GPU Communication via Fast Barrier Synchronization. In: Parallel Distributed Processing, 2010 IEEE International Symposium on. pp. 1–12. IPDPS '10 (2010)
[30] Yan, S., Long, G., Zhang, Y.: StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 229–238. PPoPP '13 (2013)