CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides

Weifeng Liu[1,2,3,*], Ang Li[4], Jonathan D. 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]*Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway*
[4]*Pacific Northwest National Lab, USA*

SUMMARY

The sparse triangular solve kernels, SpTRSV and SpTRSM, are important building blocks for a number of numerical linear algebra routines. Parallelizing SpTRSV and SpTRSM 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 overheads between the sets. To address this, we propose in this paper novel approaches for SpTRSV and SpTRSM 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. To further exploit the data-parallelism, we also develop an adaptive scheme for efficiently processing multiple right-hand sides in SpTRSM. A comparison with a state-of-the-art library supplied by the GPU vendor, using 20 sparse matrices on the latest GPU device, shows that the proposed approach obtains an average speedup of over two for SpTRSV and up to an order of magnitude speedup for SpTRSM. In addition, our method is up to two orders of magnitude faster for the preprocessing stage than existing SpTRSV and SpTRSM methods.

Received . . .

KEY WORDS: synchronization-free algorithm; sparse triangular solve; GPU; manycore processor

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 [9, 12], preconditioned iterative methods [36], and least squares problems [6]. This operation computes a dense solution vector x from a sparse linear system $Lx = b$ or $Ux = b$, where L and U are square lower and upper triangular sparse matrices, respectively, and b is a dense vector. According to the order of processing the components, the $Lx = b$ process is called *forward substitution*, and the $Ux = b$ process is known as *backward substitution*. When the same sparse triangular systems have to be solved for multiple right-hand sides, the operations become $LX = B$ and $UX = B$, where X and B are usually tall-and-skinny dense matrices. Such operations are called SpTRSM.

*Correspondence to: Niels Bohr Institute, University of Copenhagen, Denmark. E-mail: weifeng.liu@nbi.ku.dk

Compared to a dense triangular solve [14] and other sparse basic linear algebra subprograms (BLAS) [13, 25] such as sparse transposition [43], sparse matrix-vector multiplication [22, 28, 29, 48] and sparse matrix-matrix multiplication [27], the SpTRSV and SpTRSM operations are more difficult to parallelize since they are inherently sequential. This is because computing any single component x_k in a lower triangular sparse solution may depend on having first computed a subset of previous components $x_0, \ldots, x_{k-1}$. Similarly, solving x_k from an upper triangular system may require a subset of subsequent components $x_{k+1}, \ldots, x_n$ to be solved beforehand. To address the inherently sequential process, 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 is in general much better than the original sequential implementation both on CPUs [17, 33, 38, 45] and on GPUs [23, 32, 34, 41].

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 hardware and software design [7, 18, 24, 31, 35].

In this paper, we improve the synchronization-free algorithm for parallel SpTRSV described in our previous work [26] and propose a new synchronization-free algorithm with an adaptive scheme for parallel SpTRSM on GPUs. Our synchronization-free algorithms require only a light-weight preprocessing stage without set partitioning. More importantly, our method completely eliminates the runtime barrier synchronizations among sets and automatically selects appropriate parameters to obtain the best performance. By doing so, our method resolves the bottlenecks and achieves significant performance improvement.

Using 20 sparse matrices (ten of them are factorized by a sparse LU method [11]) from the University of Florida Sparse Matrix Collection [10], our SpTRSV and SpTRSM methods achieve an average speedup of over two for SpTRSV and up to an order of magnitude speedup for SpTRSM over vendor supplied parallel routines for forward and backward substitution in single and double precision. More impressively, the preprocessing stage of our algorithm is up to two orders of magnitude faster than existing set-based methods in the vendor supplied libraries.

2. BACKGROUND

2.1. Solving a Sparse Triangular System with a Single Right-Hand Side (SpTRSV)

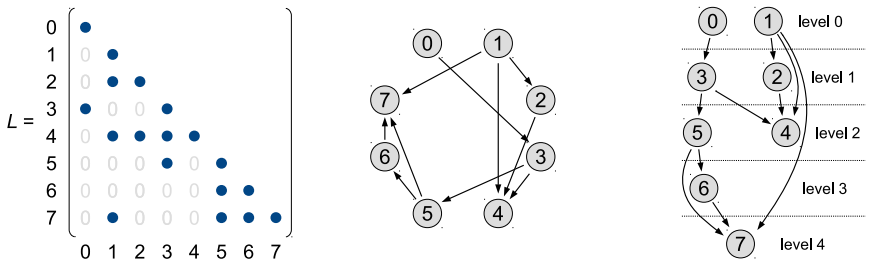
2.1.1. Serial SpTRSV Algorithm Without loss of generality, in this paper we assume that the input matrices L and U are nonsingular lower and upper triangular matrices, and are stored in the *compressed sparse column* (CSC) format consisting of three arrays `col_ptr`, `row_idx` and `val`. A typical serial forward substitution implementation of SpTRSV for solving $Lx = b$ is given in Algorithm 1. This method accesses all columns in ascending order (line 3) and solves for a single component of x at 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). Analogously, the backward substitution for solving $Ux = b$ starts from x_{n-1} and works in descending order towards x_0.

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 also update `left_sum[`i`]` (line 6). To clarify this we give an example. Figure 1 (a) shows a matrix L, for which the underlying dependencies are illustrated in its 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.

DOI: 10.1002/cpe

Algorithm 1 A serial SpTRSV method for $Lx = b$, where L is 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.

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

2.1.2. Level-Set Method for Parallel SpTRSV The motivation for 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 [37] 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.

2.1.3. Motivation for Avoiding Synchronization Synchronization remains a performance bottleneck for many applications and has long been a classic problem in computer systems research [7, 18, 24, 34, 35]. To evaluate the synchronization cost in SpTRSV, we run a parallel SpTRSV implemented by Park et al. [33] 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[†] from the University of Florida Sparse Matrix Collection [10].

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 I. Breakdown of a basic level-set method used in the source code of [33] on Intel dual-socket E5-2695 v3. See Table III for details of the four matrices.

[†]Similar to [33], the nonsingular matrix L is the lower triangular part of the input matrix, plus a dense main diagonal.

DOI: 10.1002/cpe

We have two observations from Table I. Firstly, the preprocessing stage takes much longer than a single call to SpTRSV. Specifically, the preprocessing stage is 4.39 (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 overheads for synchronization dominate 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 overheads 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 overheads for preprocessing (i.e., generating level-sets) and to avoid the runtime barrier synchronizations.

2.2. *Solving a Sparse Triangular System with Multiple Right-Hand Sides (SpTRSM)*

2.2.1. Data-Parallel SpTRSM Algorithm Compared to SpTRSV with a single right-hand side, the SpTRSM kernel has better data-level parallelism since its multiple right-hand sides can be processed in parallel. Moreover, since entries in each column are independent of each other, they can be processed in parallel as well. Algorithm 2 shows a data-level parallel method for SpTRSM. It can be seen that the main *for* loop (lines 8–14) has two optimization strategies for leveraging data-level parallelism: one is to parallelize multiple entries in each column (line 8), the other is to parallelize multiple right-hand sides (line 9).

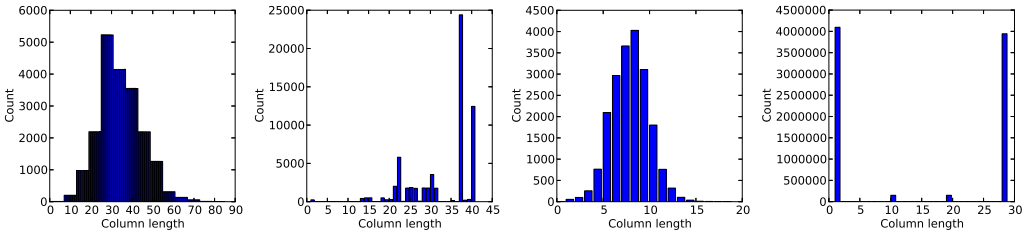
Algorithm 2 A data-parallel SpTRSM method for $LX = B$, where L is in the CSC format.

```
1: MALLOC(*left_sum, n × rhs)                          ▷ rhs is the number of right-hand sides.
2: MEMSET(*left_sum, 0)
3: for i = 0 to n − 1 in parallel do
4:     for ri = 0 to rhs − 1 do
5:         loc ← i × rhs + ri
6:         X[loc] ← (B[loc]-left_sum[loc])/val[col_ptr[i]]
7:     end for
8:     for j = col_ptr[i]+1 to col_ptr[i + 1]−1 in parallel do   ▷ Optimization 1: parallelize column entries.
9:         for ri = 0 to rhs − 1 in parallel do                  ▷ Optimization 2: parallelize right-hand sides.
10:            locls ← row_idx[j] ×rhs + ri
11:            loc ← i × rhs + ri
12:            left_sum[locls] ← left_sum[locls] + val[j] × X[loc]
13:        end for
14:    end for
15: end for
16: FREE(*left_sum)
```

2.2.2. Motivation for an Adaptive Method If one thread is used for solving a batch of components (i.e., a row of the solution matrix X), the two optimization strategies for parallelizing the two *for* loops are mutually exclusive. That is to say, a program must select either *for* loop to parallelize. Because of the different column lengths and the number of right-hand sides, directly parallelizing any one of the two *for* loops may not always achieve the best performance. Figure 2 shows the distribution of column lengths for the four matrices used above. It can be seen that the distribution varies from matrix to matrix. Some matrices only have short columns, meaning that the SIMD parallelism of column length may not be enough for saturating modern GPUs (for the NVIDIA the warp size is 32 and for the AMD the wavefront size is 64). For instance, assume that the number of right-hand sides is 16, then parallelizing the loop on right-hand sides (optimization 2 in line 9 of Algorithm 2) may give the best performance for matrix *Chipcool0* as most of its columns are shorter than 16. But this may degrade throughput for matrix *FEM/ship_003* since most of its columns are longer than 16, and in such a case parallelizing the loop on column length (optimization 1 in line 8 of Algorithm 2) will be expected to be better.

Thus, it is important to find an adaptive method that can select a *for* loop to parallelize for best data-parallelism and best throughput.

DOI: 10.1002/cpe



FEM/ship_003 FEM/Cantilever chipcool0 nlpkkt160

Figure 2. The distribution of column length (the number of columns as y axes) of four matrices.

3. SYNCHRONIZATION-FREE ALGORITHM FOR SPTRSV

The first objective of this work is to eliminate the cost of 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 respect them when solving at runtime.

In this work, we use GPUs as the platform for exploiting 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 need 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 in forward substitution, see Figure 1 (b), and vice versa in backward substitution), we can map entries to warps and strictly respect the partial order of the entries so that no warp execution deadlock will occur.

Therefore, before 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 [43]. Compared to 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 3 show the pseudocode for our preprocessing stage.

Knowing the in-degree information indicating how many warps have to be finished in advance, we can initiate a sufficient number of warps to fully exploit the irregular parallelism. For an arbitrary warp, after finishing the necessary floating-point computation for a component (line 14 in Algorithm 3), 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.

DOI: 10.1002/cpe

Algorithm 3 The proposed synchronization-free algorithm for SpTRSV (forward substitution).

```
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-INCR(&d_in_degree[row_idx[i]])
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 4.
18:                ATOMIC-ADD(&s_left_sum[rid], val[j] × x[i])
19:                ATOMIC-INCR(&s_in_degree[rid])
20:            else                                        ▷ Use GPU off-chip memory for green area in Figure 4.
21:                ATOMIC-ADD(&d_left_sum[rid], val[j] × x[i])
22:                ATOMIC-DECR(&d_in_degree[rid])
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)
```

Figure 3 illustrates the procedure for our synchronization-free algorithm[‡] 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*.

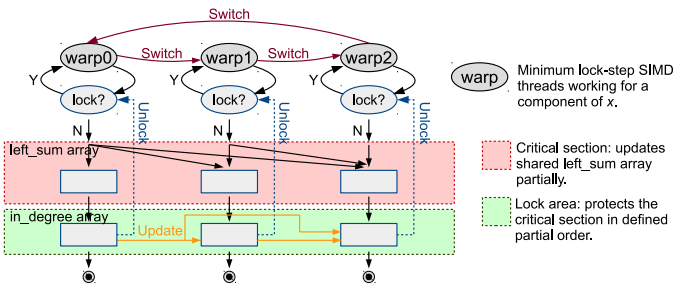


Figure 3. The basic procedure of our synchronization-free algorithm.

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 NVIDIA nvcc compiler would not start the waiting loop for some 'optimization' reasons, so a signal will be sent to the hardware warp scheduler to switch to the next warp context. This avoids the execution deadlock. In contrast, the AMD OpenCL compiler does not have this risk at all, so

[‡]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.

DOI: 10.1002/cpe

the waiting loop in our OpenCL version does not have to add any functions to prevent deadlock. In the **critical section** phase, the warp updates the components in `left_sum` that have dependencies on the components that 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 `left_sum` (so that all the order dependencies are strictly respected). Depending on the number of components in that column (line 15 in Algorithm 3), 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 3 give the pseudocode for the solving stage of our synchronization-free SpTRSV method. We can optimize this by exploiting the GPU on-chip scratchpad memory. When warps in the same thread-block share the same portion of on-chip memory, some components' dependencies may be resolved within the thread block with lower latency. Our implementation allocates two sets of intermediate arrays, one set on local scratchpad memory (`s_left_sum` and `s_in_degree`) and the other set on GPU off-chip global memory (`d_left_sum` and `d_in_degree`), see line 1 of Algorithm 3. When a warp finds a dependent entry (the later entry that depends on the current one) 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 satisfied.

Figure 4 (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 3), other operations in GPU off-chip memory are marked green (lines 21–22), and the diagonal entries are coloured blue (line 14). Figure 4 (b) plots read/write behaviour 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 blue arrow blocks for columns 0, 1 and 5 in 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 (see red and green arrow blocks in the top half) until their in-degrees are eliminated for solving (see blue arrow blocks). Figure 5 plots an example that solves an upper triangular system where the matrix is the symmetric counterpart of the lower triangular matrix shown in Figure 4. It can be seen that even though the two matrices are symmetric to each other, the two SpTRSV processes have completely different parallelism.

Also, it can be seen that if the entries in a given column (except the one on the diagonal) are sorted in ascending order, the components affected by the column will be expected to be solved in ascending order (i.e., from left to right). Because the left components are in general likely to finish earlier and thus signal their out-degree components earlier, the overall waiting time of our synchronization-free algorithm may be decreased and better performance can be expected. As for backward substitution, the column entries can be accessed in reverse order for similar effects. In this procedure, a fast segment sort [15] that separately orders a list of columns in parallel will be important to achieve overall best performance.

4. SYNCHRONIZATION-FREE ALGORITHM FOR SPTRSM

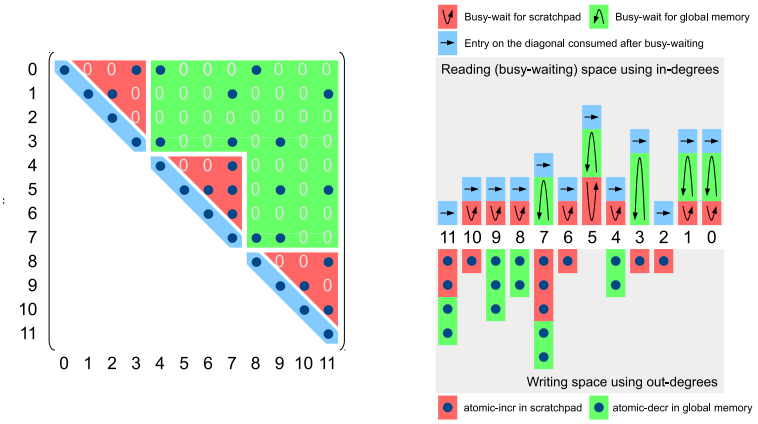
Based on the SpTRSV approach described in the previous section, we develop an extended synchronization-free algorithm for SpTRSM. The key idea is to adaptively select an optimization strategy (i.e., parallelizing multiple nonzero entries in a column, or parallelizing multiple right-hand sides) at runtime for each column to achieve best performance.

This adaptive method can be illustrated in a two-dimensional space shown in Figure 6. The two dimensions are the length of a given column (the number of nonzero entries in the column, expressed as col_len, i.e., 'column length', on the x axis) and the number of right-hand sides (expressed as

DOI: 10.1002/cpe

(a) A matrix L. (b) Read/write behaviours.

Figure 4. An example of the proposed synchronization-free SpTRSV method for forward substitution. The red area performs atomic-add (line 18 in Algorithm 3) and atomic-incr (line 19) in scratchpad memory, and the green area performs atomic-add (line 21) and atomic-decr (line 22) in GPU global memory.



(a) A matrix U. (b) Read/write behaviours.

Figure 5. An example of the proposed synchronization-free SpTRSV method for backward substitution.

rhs, i.e., '#right hand sides', on the y axis). Two parameters p and q are used for partitioning the space vertically and horizontally, respectively. Hence the space is divided into four areas utilizing different optimization strategies to process a column:

- The upper right and lower right parts are always processed using the optimization method 1. This means that when col_len is long enough (it may be much larger than rhs), parallelizing multiple nonzero entries in the column will give better performance.
- The upper left part is always processed by the optimization strategy 2, meaning that parallelizing multiple right-hand sides will offer superior performance when rhs is sufficiently large and col_len is short enough. Moreover, compared to parallelizing multiple nonzero entries that causes all subsequent components released at roughly the same time, parallelizing multiple right-hand sides processes the components as early as possible, thus potentially reducing the overall execution time.
- The lower left area is further divided into two parts: the left part indicates that when rhs is larger than col_len, optimization method 2 should be used, and the right part indicates that when col_len is long enough but still under the vertical divider p, the column should be processed by optimization method 1.

DOI: 10.1002/cpe

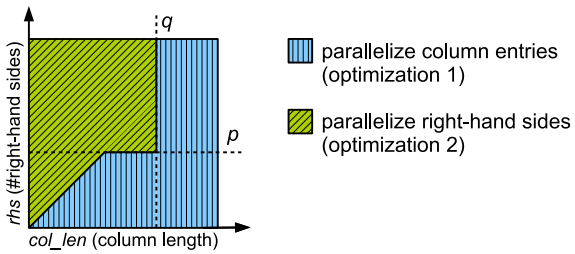


Figure 6. The adaptive method for computation in each column.

Algorithm 4 The proposed synchronization-free algorithm for SpTRSM (forward substitution).

```
1: MALLOC(*d_left_sum, n × rhs)                          ▷ rhs is the number of right-hand sides.
2: MALLOC(*d_in_degree, n)
3: MEMSET(*d_left_sum, *d_in_degree, 0)
4: function PREPROCESSING-STAGE()                        ▷ The same to the preprocessing stage in SpTRSV.
5:     for i = 0 to nnz − 1 in parallel do
6:         ATOMIC-INCR(&d_in_degree[row_idx[i]])
7:     end for
8: end function
9: function SOLVING-STAGE()
10:    for i = 0 to n − 1 in parallel do                 ▷ One concurrent warp for one component.
11:        while d_in_degree[i] ≠ 1 do
12:            //busy wait
13:        end while
14:        for ri = 0 to rhs − 1 in parallel do          ▷ One thread for one right-hand side item.
15:            loc ← i × rhs + ri
16:            X[loc] ← (B[loc]-d_left_sum[loc])/val[col_ptr[i]]
17:        end for
18:        len ← col_ptr[i + 1] − col_ptr[i]−1
19:        if (len ≤ rhs ∨ rhs > p) ∧ len < q then       ▷ In this paper, p and q are set to 8 and 8192, respectively.
20:            for j = col_ptr[i]+1 to col_ptr[i + 1]−1 do
21:                rid ← row_idx[j]
22:                for ri = 0 to rhs − 1 in parallel do  ▷ One thread for one right-hand side item.
23:                    ATOMIC-ADD(&d_left_sum[rid × rhs + ri], val[j] × X[i × rhs + ri])
24:                    ATOMIC-DECR(&d_in_degree[rid])
25:                end for
26:            end for
27:        else
28:            for j = col_ptr[i]+1 to col_ptr[i + 1]−1 in parallel do   ▷ One thread for one nonzero.
29:                rid ← row_idx[j]
30:                for ri = 0 to rhs − 1 do
31:                    ATOMIC-ADD(&d_left_sum[rid × rhs + ri], val[j] × X[i × rhs + ri])
32:                    ATOMIC-DECR(&d_in_degree[rid])
33:                end for
34:            end for
35:        end if
36:    end for
37: end function
38: FREE(*d_left_sum, *d_in_degree)
```

Here we do not adopt the optimization that uses both on-chip and off-chip memory for faster solving stage in our synchronization-free SpTRSV. The main reason is that the number of right-hand sides can be large and thus may consume excessive on-chip scratchpad memory and degrade device occupancy (i.e., cannot saturate hardware resources). As a comparison, the proposed algorithm for SpTRSM stores shared resources, such as the `in-degree` and `left_sum` arrays, only in GPU off-chip global memory. Algorithm 4 shows the proposed synchronization-free algorithm for forward substitution SpTRSM. It can be seen that line 19 decides which optimization approach is called for

DOI: 10.1002/cpe

a given column. In our experiments, we always set p and q to 8 and 8192, respectively, since we find empirically that this combination generally gives the best overall performance (see Figure 12).

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

We have implemented the proposed synchronization-free SpTRSV method both in CUDA and in OpenCL and have evaluated it on two GPUs: an NVIDIA GeForce Titan X GPU of Pascal architecture, and an AMD Radeon R9 Fury X GPU of GCN architecture. We also benchmark the most recent SpTRSV and SpTRSM implementations from two libraries cuSPARSE v8.0 and MKL v11.3 Update 3 provided by NVIDIA and Intel, respectively. We execute each method a hundred times and use the arithmetic average of the runtime to calculate the GFlop/s rates.

Because mixed-precision numerical methods have recently attracted much attention, we evaluate all methods in both single and double precision for SpTRSV. But for brevity, we only show experimental results of SpTRSM in double precision. Information about the platforms and test schemes are listed in Table II.

The testbeds	The SpTRSV and SpTRSM algorithms
An Intel Xeon E5-2630 v4 (Broadwell, 10 cores @ 2.2 GHz, 25 MB L3 cache, 16 GB DDR4 @ 68.3 GB/s).	(1) The `mkl_?csrtrsv` and `mkl_?csrsm` for SpTRSV and SpTRSM in MKL v11.3 Update 3, respectively. Note that those two are 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 for SpTRSV in MKL v11.3 Update 3. (3) The parallel executor `mkl_sparse_?_trsm` using the functions `mkl_sparse_set_sm_hint` and `mkl_sparse_optimize` as an inspector for SpTRSM in MKL v11.3 Update 3.
An NVIDIA GeForce Titan X (Pascal GP102, 3584 CUDA cores @ 1.4 GHz, 12 GB GDDR5X @ 480 GB/s, driver v367.48).	(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 v8.0. (2) The latest SpTRSM method `cusparse?csrsm_solve` using function `cusparse?csrsm_analysis` in its preprocessing stage in the NVIDIA cuSPARSE v8.0. (3) The synchronization-free methods for SpTRSV and SpTRSM 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 methods for SpTRSV and SpTRSM proposed in this paper.

Table II. The testbeds and SpTRSV and SpTRSM algorithms. Note that the proposed synchronization-free method uses matrices in CSC format, while the other methods use the CSR format, and we assume the input is already in the right format before a solve starts.

Table III lists the 20 sparse matrices used for our experiments on all platforms. The first 10 matrices have been used in other research on sparse matrix computations [17, 25, 27, 28, 29, 33, 48] and are publicly available from the University of Florida Sparse Matrix Collection [10]. The last 10 matrices are also from the collection but, in this evaluation, we use their factorized forms generated by a sparse LU decomposition using MA48 [11] from the Harwell Subroutine Library (HSL) [16]. Hence our benchmark suite covers more application scenarios of sparse system solvers. 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 of its components can run in parallel, whereas for the factorized matrix *g7jac140sc* very few

DOI: 10.1002/cpe

components can utilize parallel resources. Note that in this paper we test both forward and backward substitution. Thus the information for the L and U parts of the matrices are listed separately.

Matrix name	#Rows/#Cols	Tria.	#Nonzeros (L or U)	#Level-sets (L or U)	Parallelism (L or U)		
					minimum	average	maximum
nlpkkt160	8,345,600	L	118,931,856	2	4,096,000	4,172,800	4,249,600
		U		2	4,096,000	4,172,800	4,249,600
road_central	14,081,816	L	31,015,229	59	2	238,674	4,480,856
		U		59	1	238,674	4,330,799
road_usa	23,947,347	L	52,801,659	77	1	311,004	6,739,633
		U		77	1	311,004	8,010,032
webbase-1M	1,000,005	L	2,348,442	512	2	1,953	88,133
		U	1,757,099	514	1	1,945	926,806
wiki-Talk	2,394,385	L	3,072,221	515	1	4,649	2,256,516
		U	6,737,959	522	1	4,586	2,341,064
chipcool0	20,082	L	150,616	534	2	37	73
		U		534	1	37	78
FEM/Cantilever	62,451	L	2,034,917	2,397	1	26	244
		U		2,397	1	26	244
crankseg_1	52,804	L	5,333,507	4,056	1	13	56
		U		4,056	1	13	280
FEM/ship_003	121,728	L	4,103,881	4,367	1	27	761
		U		4,367	1	27	59
hollywood-2009	1,139,905	L	57,515,616	82,735	1	13	91,152
		U		82,735	1	13	78,886
Wordnet3	82,670	L	166,941	8	567	10,333	30,548
		U	110,482	74	1	1,117	70,543
rajat18	94,294	L	280,494	82	1	1,149	56,915
		U	460,208	167	1	564	4,150
lung2	109,460	L	273,646	454	1	241	37,338
		U	754,446	906	1	120	7,620
dc2	116,835	L	666,173	677	1	172	85,849
		U	848,612	678	1	172	41,698
soc-sign-epinions	131,828	L	271,947	28	1	4,708	87,001
		U	818,316	1,566	1	84	105,087
TSOPF_RS_b162_c4	20,374	L	1,565,005	765	3	26	5,273
		U	3,949,099	1,129	1	18	229
cit-HepTh	27,770	L	4,244,363	879	1	31	17,970
		U	2,130,249	1,191	1	23	7,918
rajat25	87,190	L	1,458,168	1,756	1	49	39,959
		U	1,928,716	1,674	1	52	572
epb3	84,617	L	3,313,794	3,470	1	24	26,906
		U	6,439,264	3,920	1	21	140
g7jac140sc	41,490	L	10,102,488	5,156	1	8	19,520
		U	17,590,850	5,327	1	7	129

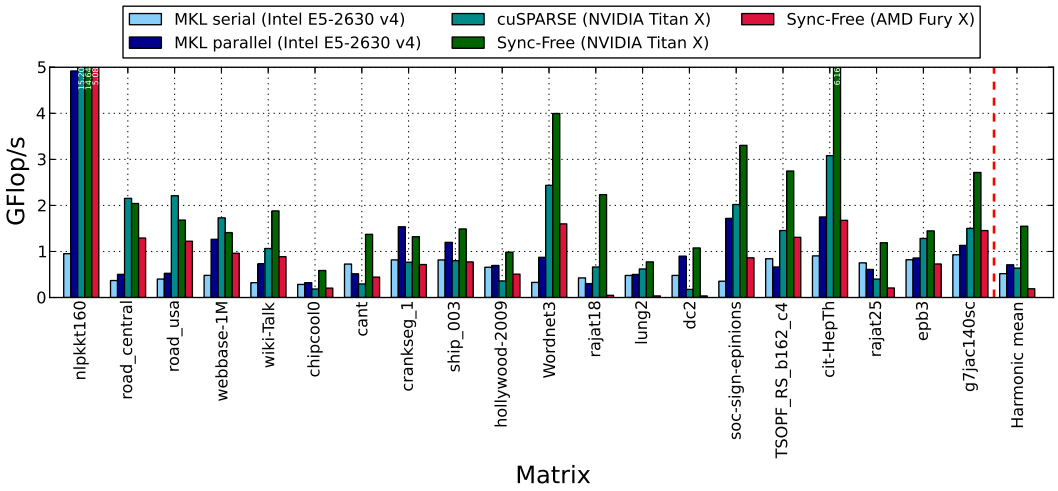
Table III. The benchmark suite including 10 original matrices (eight symmetric and two unsymmetric) and 10 matrices (all unsymmetric) factorized using a sparse LU method. Note that even for symmetric matrices, forward and backward substitution may not have the same parallelism. Also note that 'parallelism' refers to the average number of components that can be solved in parallel in the level-set method. For example, the 8×8 matrix plotted in Figure 1 has a parallelism of 1.6 (i.e., eight components grouped into five parallelizable level-sets).

5.2. *SpTRSV Performance*

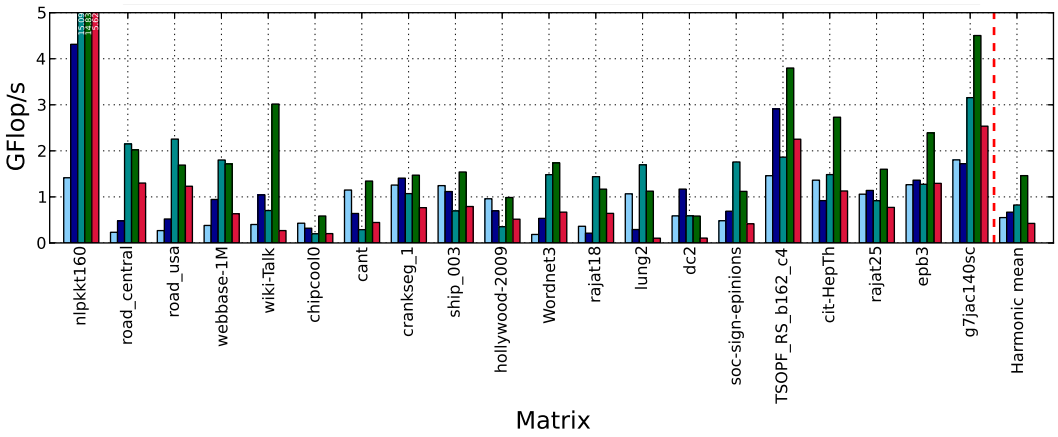
Figures 7 and 8 show the single and double precision SpTRSV performance on the 20 matrices measured on the three platforms. Overall, the methods in MKL are relatively slow when the parallel degree is high, but behave better when the parallel degree is low (meaning that operations are more sequential). The cuSPARSE library exhibits opposite performance: showing relatively better performance when the parallel degree is high but inferior performance when the parallel degree is low. Nevertheless, on average (harmonic mean), the MKL and cuSPARSE libraries show

DOI: 10.1002/cpe

comparable throughput. Compared to both MKL and cuSPARSE, our synchronization-free method is in general faster and sometimes much faster.



(a) forward substitution



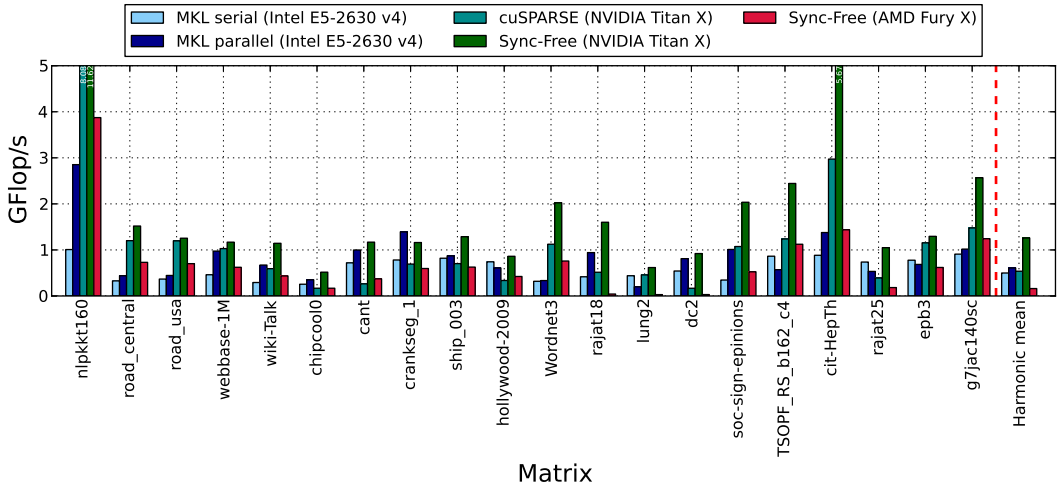
(b) backward substitution

Figure 7. The SpTRSV (single precision) performance of the 20 matrices on three platforms.

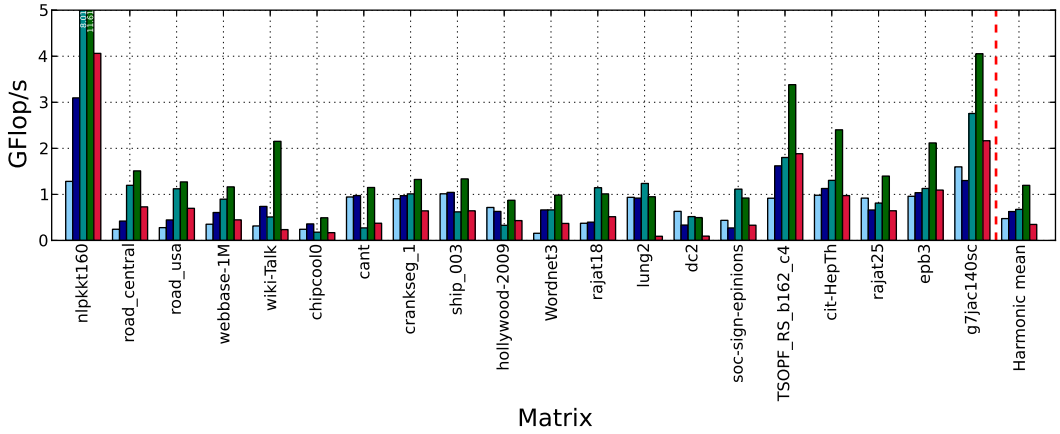
Specifically, on the Pascal-based Titan X GPU, our synchronization-free algorithm demonstrates an average speedup over the cuSPARSE library of 2.42 times in single precision and 2.34 times in double precision for forward substitution, and 1.77 times in single precision and 1.77 times in double precision for backward substitution. The maximum speedups are 6.11, 5.49, 4.65, 4.22, respectively. These best speedups are all from matrices, such as *cant* and *dc2*, that have most nonzero entries in diagonal blocks. For those matrices, the optimizing strategy of using both scratchpad and off-chip memory improves the overall performance. Also, it can be seen that our method achieves speedups of 2.73, 2.56, 2.8 and 2.65, respectively, for matrix *hollywood-2009*. This matrix requires 82,735 runtime synchronizations (see Table III) limiting its performance for the level-set methods. In contrast, our method avoids synchronizations and 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.

We also notice that compared to the Kepler- and Maxwell-based GPUs used in our previous work [27], the Pascal-based Titan X GPU offers higher performance. The major reason is that the Pascal architecture is equipped with higher bandwidth and improved micro-architectures for atomic operations, which are extensively utilized in our approach. Actually, Scogland and Feng [39]

DOI: 10.1002/cpe



(a) forward substitution



(b) backward substitution

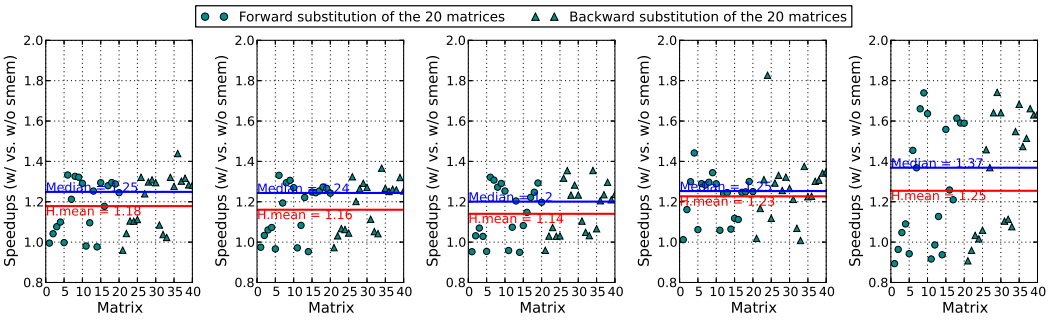
Figure 8. The SpTRSV (double precision) performance of the 20 matrices on three platforms.

also confirmed that atomic operations have been continuously improved in the latest generations of modern GPUs. Moreover, although the AMD Fury X GPU has slightly higher bandwidth than the NVIDIA Titan X, it is in general slower for our synchronization-free SpTRSV algorithm. The main reason is probably the implementation differences for warp/wavefront scheduling in the two vendor's products.

To further measure the impact of using the on-chip memory for lower latency, we list the performance improvement obtained by this optimization technique in Figure 9. Because an NVIDIA CUDA thread block allows up to 1024 threads (i.e., 32 warps of 32 threads), we report different thread block configurations of 4-, 8-, 16- and 32-warps in Figures 9 (a)–(d), respectively. In contrast, OpenCL on AMD cards can use up to 256 threads (i.e., four wavefronts of 64 threads). Thus we only test performance of thread block of four wavefronts and plot it in Figure 9 (e). In each subfigure, the forward and backward substitution of the 20 test matrices are listed (see 40 sample points on the x axis), and the median and harmonic mean values of the speedups are highlighted. As can be seen, the on-chip memory optimization technique yields higher performance in most of the cases (i.e., with speedups higher than 1.0). Specifically, the 32-warp setting (Figure 9 (d)) yields the highest speedups since its diagonal blocks (recall red areas in Figures 4 and 5) are larger than those in other settings. However, the absolute throughput of the 32-warp setting (not shown here for brevity) is in general a bit slower than the other three with very similar performance. So in this paper we always

DOI: 10.1002/cpe

set the number of warps to 16. As for the AMD Fury X, we also notice obvious speedups from the on-chip memory optimization. But because of the limited combinations, we use the 4-wavefront configuration in our test.



(a) Titan X (4-warp) (b) Titan X (8-warp) (c) Titan X (16-warp) (d) Titan X (32-warp) (e) Fury X (4-wavefront)

Figure 9. The impact of using the on-chip memory for better performance. The y axis shows speedups with this optimization technique, and the x axis shows forward and backward substitution of the 20 test matrices. The median and harmonic mean of the speedups are highlighted.

5.3. SpTRSM Performance

In this paper, we only show forward and backward substitution SpTRSM in double precision for brevity. Figures 10 and 11 show the performance of the serial/parallel methods in vendor libraries MKL and cuSPARSE, and three optimization strategies: (1) `Sync-free opt1` (parallelizing column entries in our synchronization-free method), (2) `Sync-free opt2` (parallelizing right-hand sides in our synchronization-free method), and (3) `Sync-free adaptive` (our synchronization-free method with the adaptive parameter selection illustrated in Figure 6 and described in Algorithm 4) running on NVIDIA and AMD GPUs.

It can be seen that our synchronization-free method is in general much faster than cuSPARSE, especially when the number of right-hand sides is large. This trend is the same as that shown for SpTRSV. However, there are two exceptions where cuSPARSE behaves better. One is for matrix *nlpkkt160*, where its parallelism is much higher than other matrices, and so cuSPARSE outperforms our method. But since cuSPARSE requires more space to save level-set information, it cannot process as many right-hand sides as the synchronization-free algorithm can (specifically, cuSPARSE does not work for more than 8 right-hand sides in our test). Another exception is from the backward substitution of matrices *rajat18* an *dc2*, where the row/column lengths of their upper triangular part are distributed in a power-law fashion (i.e., several are of size $O(n)$ and the rest are of size $O(1)$). In this case, the in-degree of some components is relatively high thus causing more load unbalanced traffic for the atomic operations in our method. As a result, the performance of the cuSPARSE method is relatively higher. Nevertheless, for most of the cases, our synchronization-free algorithm achieves significant speedups (up to a few tens) over MKL and cuSPARSE. In addition, cuSPARSE fails to solve both the L and U parts of matrices *road_central* and *road_usa*, maybe due to its complex approach analysing their sparsity structures. It can also be seen that the speedups in SpTRSM are more noticeable than in the SpTRSV test. The reason may be that cuSPARSE does not select the best parallel scheme for multiple right-hand sides.

We can further see that any one of the two optimization methods (parallelizing column elements or right-hand sides) does not always behave the best, but our adaptive strategy outperforms both in most cases. For instance, although the optimization method 2 commonly outperforms the method 1, matrices *nlpkkt160* and *webbase-1M* actually show that the method 1 can be much faster than the method 2. In contrast, our adaptive scheme almost always behaves faster than both optimization methods 1 and 2 on the two matrices. In addition, for the matrices preferring optimization method 2, our adaptive strategy can almost always offer identical or better performance.

DOI: 10.1002/cpe

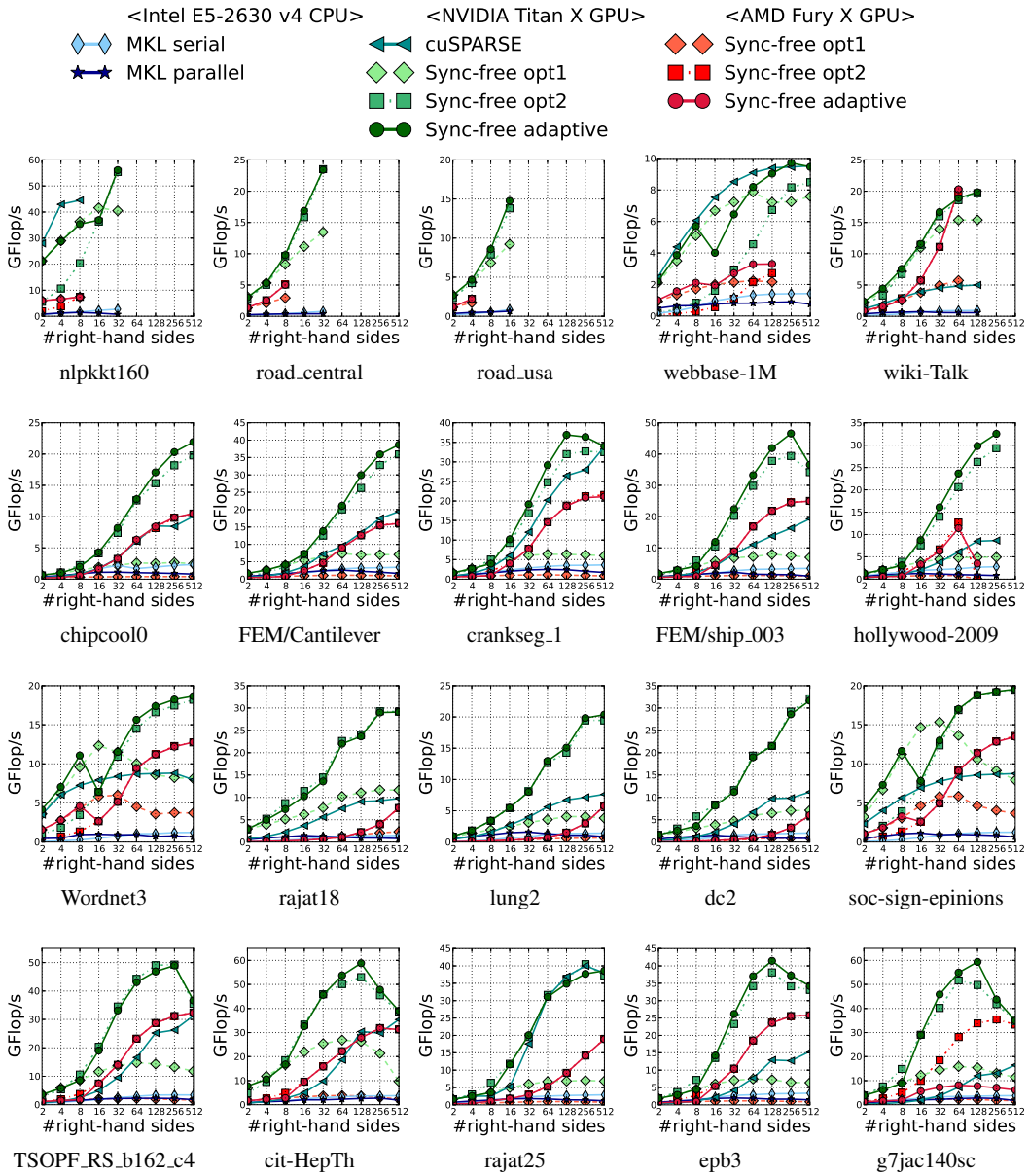


Figure 10. The SpTRSM (forward substitution) performance of the 20 matrices on three platforms. Note that some numbers of right-hand sides are not tested because of the memory capacity limitations.

We also test the impact of changing the parameter p. Figure 12 shows relative performance of setting $p = 4, 8, 16, 32$ and 64 over $p = 4$ on the Titan X and Fury X platforms. Note that the relative performance is calculated as the harmonic mean of speedups between forward/backward substitutions for the 20 test matrices. It can be seen that setting p to 8 in general brings the best performance for all cases, in particular for the case of 16 right-hand sides. Hence in this work we always set p to the fixed value 8.

On the other hand, because the parameter q is a split point for column length (recall Figure 6), setting a proper value for q depends strongly on the test matrices selected. In our benchmark suite, we fix p to 8 and extensively test SpTRSM with $q = 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192$ and 16384 and empirically set q to 8192 for the best observed throughput. In fact the majority

DOI: 10.1002/cpe

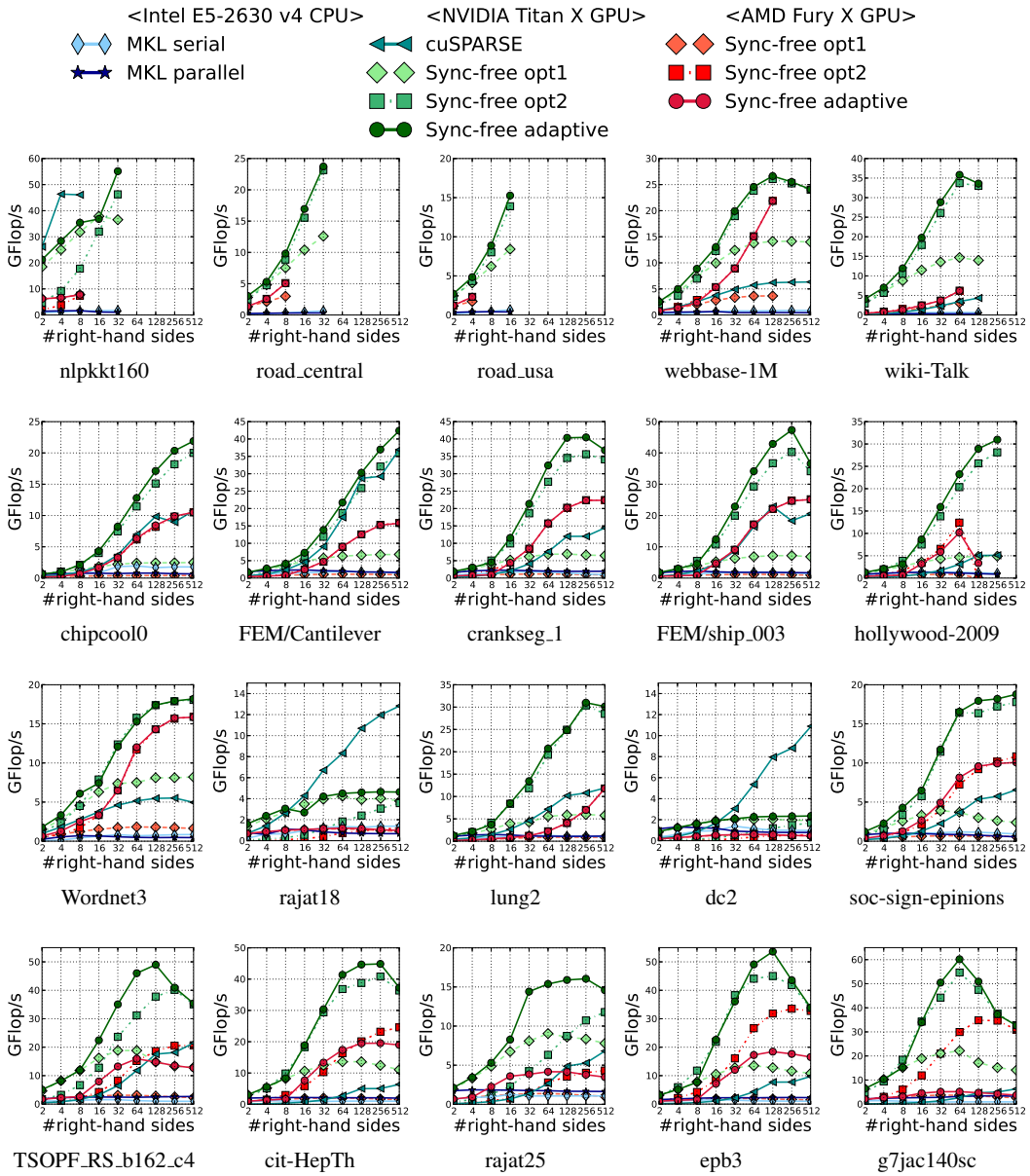
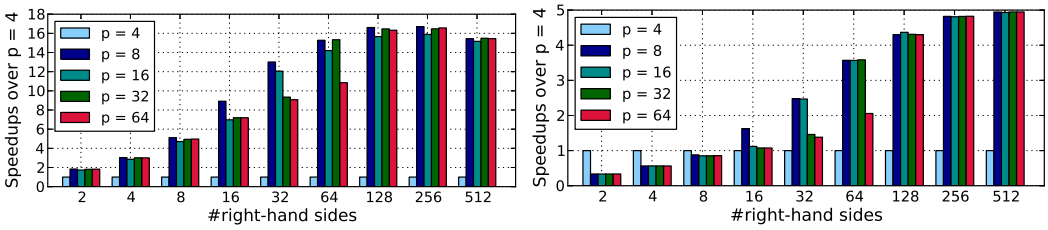


Figure 11. The SpTRSM (backward substitution) performance of the 20 matrices on three platforms.

of the matrices (specifically, 19 out of 20) do not have columns with more than a few tens of nonzeros, meaning that they are insensitive to a larger q, thus no noticeable performance difference is observed. The exception is the factorized matrix *g7jac140sc*, which is much denser than the other test matrices (see Table III). In this case, setting q to values larger than 1024 gradually improves performance, since this scheme exploits more parallelism over the right-hand sides for updating subsequent components with lower latency.

It is also worth noting that tuning parameters in general needs further extensive benchmarking with more combinations of parameters p, q for more test matrices. As can be seen, the parameter tuning method presented in this paper is a reasonable initial attempt, and we leave more flexible and efficient autotuning as future work.

DOI: 10.1002/cpe



(a) Titan X. (b) Fury X.

Figure 12. The impact of setting p to different values $4, 8, 16, 32$ and 64. Note that the y axis is the relative performance of all settings over $p = 4$. Note that q is set to 8192 for all tests.

5.4. Overhead for Preprocessing

Tables IV and V show the preprocessing overheads of the parallel forward and backward SpTRSV and SpTRSM implementations from MKL, cuSPARSE and our approach on the three platforms. As can be seen, our method achieves an average speedup of over 48.4 (maximum of 121, from backward substitution of matrix *chipcool0*) over the SpTRSV method in the cuSPARSE library on the Titan X card. For the SpTRSM operation, the speedups are on average over 322.9 with a maximum of 690 (from backward substitution of matrix *cit-HepTh*). 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 comparable cost for preprocessing, due to similar off-chip memory bandwidth.

Matrix name	Forward substitution				
	Intel E5-2630 v4	NVIDIA Titan X (Pascal)			AMD Fury X
	MKL v11.3 U3	cuSPARSE v8.0		Sync-Free	Sync-Free
		SpTRSV	SpTRSM		
nlpkkt160	2675.80	12.18	479.32	5.19	4.95
road_central	3735.04	27.21	Failed	5.44	5.68
road_usa	6190.39	55.82	Failed	2.69	2.33
webbase-1M	351.72	3.86	23.76	0.10	0.12
wiki-Talk	783.44	7.27	39.39	0.17	0.16
chipcool0	14.68	1.18	6.90	0.01	0.03
FEM/Cantilever	78.70	5.19	24.09	0.08	0.09
crankseg_1	125.70	4.13	45.94	0.23	0.20
FEM/ship_003	196.49	3.14	44.76	0.15	0.16
hollywood-2009	14116.38	148.04	919.10	3.25	2.86
Wordnet3	23.9	0.57	4.12	0.01	0.01
rajat18	34.85	1.11	5.71	0.03	0.02
lung2	56.92	1.30	8.83	0.05	0.08
dc2	55.73	6.68	14.13	0.13	0.07
soc-sign-epinions	88.32	0.89	5.17	0.01	0.06
TSOPF_RS_b162_c4	78.23	0.98	11.76	0.10	0.12
cit-HepTh	68.68	3.29	23.81	0.13	0.08
rajat25	83.03	5.55	25.24	0.08	0.07
epb3	212.49	3.21	44.09	0.18	0.20
g7jac140sc	517.86	7.94	107.86	0.38	0.53
Harmonic mean	76.26	2.28	13.90	0.05	0.07

Table IV. Preprocessing cost (in millisecond) of the tested methods for forward substitution on three devices. Note that the SpTRSV and SpTRSM routines in cuSPARSE do not use the same strategies for preprocessing and thus have different costs.

6. RELATED WORK

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

DOI: 10.1002/cpe

Matrix name	Backward substitution				
	Intel E5-2630 v4	NVIDIA Titan X (Pascal)			AMD Fury X
	MKL v11.3 U3	cuSPARSE v8.0		Sync-Free	Sync-Free
		SpTRSV	SpTRSM		
nlpkkt160	2624.59	12.34	492.49	5.17	4.33
road_central	3770.27	27.51	Failed	5.35	5.64
road_usa	6218.89	56.32	Failed	2.72	2.35
webbase-1M	336.78	3.32	22.42	0.14	0.11
wiki-Talk	778.67	16.11	49.36	0.55	0.55
chipcool0	15.02	1.21	7.18	0.01	0.03
FEM/Cantilever	83.78	4.30	25.59	0.08	0.10
crankseg_1	124.02	4.07	45.05	0.28	0.20
FEM/ship_003	167.07	3.21	46.85	0.14	0.16
hollywood-2009	13781.93	153.73	938.82	3.33	2.79
Wordnet3	19.68	0.82	3.83	0.01	0.01
rajat18	26.02	0.92	7.78	0.02	0.04
lung2	37.94	1.61	12.76	0.06	0.07
dc2	54.30	2.27	17.60	0.08	0.13
soc-sign-epinions	33.67	1.64	18.57	0.04	0.02
TSOPF_RS_b162_c4	35.50	1.66	22.15	0.22	0.05
cit-HepTh	101.71	2.18	89.89	0.10	0.14
rajat25	75.28	3.26	29.26	0.09	0.09
epb3	126.24	5.41	71.13	0.23	0.12
g7jac140sc	304.55	6.64	176.20	0.61	0.32
Harmonic mean	61.75	2.56	18.39	0.05	0.06

Table V. Preprocessing cost (in millisecond) of the tested methods for backward substitution on three devices.

Anderson and Saad [1] and Saltz [37] proposed that **level-sets** can expose parallelism in sparse triangular solves. A few recently developed parallel SpTRSV implementations have improved the level-set method for better data locality and faster synchronization [17, 33, 45]. Maumov [32] implemented a level-set method on NVIDIA GPUs with a tradeoff for decreasing the number of synchronizations. Li and Saad [23] 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 and SpTRSM algorithms do not analyse the sparsity structure of the input matrix and thus completely avoid the costs for generating sets and executing barrier synchronization. As a result, our method in general shows much better performance than level-set methods.

Schreiber and Tang [38] 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. [41] 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. Picciau et al. [34] recently proposed a method that partitions the graph form of an input matrix into multiple sub-graphs to obtain better data locality and higher concurrency. However, its pre-processing cost can be even more expensive than colour-set approaches.

There are also several classes of methods that do not create sets in advance. Mayer [30] pointed out that **2D decomposition** can accelerate SpTRSV but needs to reorganize the data structure of the input matrix. Smith and Zhang [40] and Totoni et al. [42] reported that reformatting the input matrix can bring higher performance. Chow and Patel [8] and Anzt et al. [2, 4, 3, 5] 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 and SpTRSM in sparse direct solvers. For the tridiagonal case, another non-trivial inherently sequential problem, there are very specific fast algorithms [44]. In contrast, the

method we have proposed 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 [46], prefix-sum scan [47], wavefront [18], sparse transposition [43], and sparse matrix-vector multiplication [22, 28, 29, 48]. 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. Moreover, we recently noticed that bypassing caches [19], improving thread-groups locality by clustering [21] and utilizing on-package high bandwidth memory [20] can further improve algorithm performance. We leave this extension as future work.

7. CONCLUSIONS

In this paper, we have proposed synchronization-free algorithms for parallel SpTRSV and SpTRSM. These methods completely eliminate the overheads for generating level-sets or colour-sets (in the preprocessing stage) and for explicit runtime barrier synchronization (in the solving stage). Meanwhile, they adaptively select optimization paths for best parallelism in the case of multiple right-hand sides. Experimental results show that our approach makes preprocessing up to two orders of magnitude faster than level-set methods, and demonstrates significant speedups over vendor supplied parallel routines for forward and backward SpTRSV and SpTRSM in single and double precision.

ACKNOWLEDGEMENT

The authors would like to thank our anonymous reviewers from *Euro-Par 2016* and *Concurrency and Computation: Practice and Experience* for their invaluable feedback. We also thank Shuai Che for helpful discussion about OpenCL programming. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the NLAFET project (grant number 671633) and the Marie Sklodowska-Curie TICOH project (grant number 752321). This research is also partially supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under the CENATE project (award number 66150). The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

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. Anzt, H., Chow, E., Huckle, T., Dongarra, J.: Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. In: 2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA). pp. 49–56 (2016)
4. Anzt, H., Chow, E., Szyld, D.B., Dongarra, J.: Domain Overlap for Iterative Sparse Triangular Solves on GPUs. In: Software for Exascale Computing - SPPEXA 2013-2015. pp. 527–545. Springer International Publishing (2016)
5. Anzt, H., Huckle, T., Bräckle, J., Dongarra, J.: Incomplete Sparse Approximate Inverses for Parallel Preconditioning. Parallel Computing (2017)
6. Björck, Å.: Numerical Methods for Least Squares Problems. Society for Industrial and Applied Mathematics (1996)
7. Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 13:1–13:11. SC '11 (2011)
8. Chow, E., Patel, A.: Fine-Grained Parallel Incomplete LU Factorization. SIAM Journal on Scientific Computing 37(2), C169–C193 (2015)
9. Davis, T.: Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics (2006)
10. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (dec 2011)
11. Duff, I.S., Reid, J.K.: The Design of MA48: A Code for the Direct Solution of Sparse Unsymmetric Linear Systems of Equations. ACM Trans. Math. Softw. 22(2), 187–226 (1996)
12. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Inc., 2nd edn. (2017)

DOI: 10.1002/cpe

13. 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)
14. Hogg, J.D.: A Fast Dense Triangular Solve in CUDA. SIAM Journal on Scientific Computing 35(3), C303–C322 (2013)
15. Hou, K., Liu, W., Wang, H., Feng, W.c.: Fast Segmented Sort on GPUs. In: Proceedings of the 31st ACM International Conference on Supercomputing. ICS '17 (2017)
16. HSL: Hsl: A collection of fortran codes for large scale scientific computation. Tech. rep., http://www.hsl.rl.ac.uk/ (2002)
17. 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)
18. 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)
19. Li, A., van den Braak, G.J., Kumar, A., Corporaal, H.: Adaptive and Transparent Cache Bypassing for GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 17:1–17:12. SC '15 (2015)
20. Li, A., Liu, W., Kristensen, M.R.B., Vinter, B., Wang, H., Hou, K., Marquez, A., Song, S.L.: Exploring And Analyzing the Real Impact of Modern On-Package Memory on HPC Scientific Kernels. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17 (2017)
21. Li, A., Song, S.L., Liu, W., Liu, X., Kumar, A., Corporaal, H.: Locality-Aware CTA Clustering For Modern GPUs. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '17 (2017)
22. Li, J., Tan, G., Chen, M., Sun, N.: SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 117–126. PLDI '13 (2013)
23. Li, R., Saad, Y.: GPU-Accelerated Preconditioned Iterative Linear Solvers. The Journal of Supercomputing 63(2), 443–466 (2013)
24. 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)
25. Liu, W.: Parallel and Scalable Sparse Basic Linear Algebra Subprograms. Ph.D. thesis, University of Copenhagen (2015)
26. Liu, W., Li, A., Hogg, J., Duff, I.S., Vinter, B.: A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In: Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings. pp. 617–630 (2016)
27. 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)
28. 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)
29. Liu, W., Vinter, B.: Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. Parallel Computing 49, 179–193 (2015)
30. Mayer, J.: Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices. Computing 86(4), 291–312 (2009)
31. Morrison, A.: Scaling Synchronization in Multicore Programs. Commun. ACM 59(11), 44–51 (2016)
32. Naumov, M.: Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Tech. rep., NVIDIA (2011)
33. 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)
34. Picciau, A., Inggs, G.E., Wickerson, J., Kerrigan, E.C., Constantinides, G.A.: Balancing Locality and Concurrency: Solving Sparse Triangular Systems on GPUs. In: Proc. IEEE Int. Conf. on High Performance Computing, Data, and Analytics (HiPC '16) (2016)
35. 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)
36. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edn. (2003)
37. Saltz, J.H.: Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. SIAM Journal on Scientific and Statistical Computing 11(1), 123–144 (1990)
38. Schreiber, R., Tang, W.P.: Vectorizing the Conjugate Gradient Method. In: Proceedings of the Symposium on CYBER 205 Applications (1982)
39. 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)
40. Smith, B., Zhang, H.: Sparse Triangular Solves for ILU Revisited: Data Layout Crucial to Better Performance. International Journal of High Performance Computing Applications 25(4), 386–391 (2011)
41. 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)

DOI: 10.1002/cpe

42. Totoni, E., Heath, M.T., Kale, L.V.: Structure-Adaptive Parallel Solution of Sparse Triangular Linear Systems. Parallel Computing 40(9), 454–470 (2014)
43. 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. pp. 33:1–33:13. ICS '16 (2016)
44. Wang, X., Xue, W., Zhai, J., Xu, Y., Zheng, W., Lin, H.: A Fast Tridiagonal Solver for Intel MIC Architecture. In: 2016 IEEE International Parallel and Distributed Processing Symposium. pp. 172–181. IPDPS '16 (2016)
45. 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)
46. 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)
47. 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)
48. Zhang, Y., Li, S., Yan, S., Zhou, H.: A Cross-Platform SpMV Framework on Many-Core Architectures. ACM Trans. Archit. Code Optim. 13(4), 33:1–33:25 (2016)

DOI: 10.1002/cpe