# Efficient Parallel Implementations of Sparse Triangular Solves for GPU Architectures

Ruipeng Li* Chaoyu Zhang†

**Abstract**

The sparse triangular matrix solve (SpTrSV) is an important computation kernel that is demanded by a variety of numerical methods such as the Gauss-Seidel iterations. However, developing efficient parallel algorithms for SpTrSV that are suitable for GPUs remains a challenging task due to the inherently sequential nature in the solve. In this paper, we revisit this problem by reviewing several parallel algorithms based on different task scheduling and different sparse matrix storage schemes, proposing modifications to the existing methods that can greatly improve the performance, and describing the implementations in details. Numerical results of Gauss-Seidel iterations with structured and unstructured matrices make evident the superiority of the proposed algorithms and implementations comparing with state-of-the-art methods in the literature.

## 1 Introduction

The prevalent many-core architectures are able to deliver enormous raw processing power in the form of massive single-instruction-multiple-data (SIMD) parallelism, where the graphics processing unit (GPU) is one of the several available platforms, which has been increasingly exploited as general-purpose processors for scientific computing. The potential of GPUs for sparse matrix computations was recognized in the early 2000s when shading languages were still required for the programming, see, e.g., [12,23]. Since the advent of CUDA, the NVIDIA GPUs have drawn much more attention for accelerating sparse matrix computations such as sparse linear solvers [13, 16, 19, 25, 35, 39–41, 44, 46] and eigensolvers [6, 8, 18, 26], where the significant performance enhancements were often achieved from the accelerated computation kernels such as sparse matrix-vector products [7, 9, 11, 14, 31, 32] and sparse matrix-matrix products [10, 30, 33]. However, the sparse triangular matrix solve (SpTrSV) remains a challenging task on GPUs due to its inherently sequential nature, and the performance reached by this kernel is usually much lower compared with the multiplication kernels. SpTrSV is demanded by a variety of numerical algorithms such as sparse direct solvers for linear systems, the Gauss-Seidel iterations, and incomplete LU (ILU) factorization preconditioners.

In this paper, we revisit this problem by reviewing the existing algorithms based on two different scheduling approaches. The first scheduling approach is level-based, where the unknowns are grouped into levels such that the unknowns within the same level can be solved simultaneously, and synchronizations are required to resolve the dependencies between the levels. To the best of our knowledge, the first level-scheduling algorithm for SpTrSV was due to Anderson and Saad in [4], and later by Saltz in [43]. The works in [25, 34] might be the first efforts on the GPU implementation and similar approaches were later adopted in [38, 45], where the sparse triangular matrix is assumed to be stored in the compressed sparse row (CSR) format. The second scheduling approach is element-based, which is more aggressive and fine-grained, where an unknown can be immediately solved after the solutions of all the unknowns that it depends on are available. The idea of element-based scheduling was first introduced in [20] for shared-memory machines, and the recent implementations on GPUs can be found in [27, 29], which assumes the compressed sparse column (CSC) format. Since synchronization is typically not required, this approach is also referred to as the synchronization-free approach. After that, we propose modifications to the existing element-scheduling algorithms that incorporates the level information, which turned out to be able to greatly improve the performance. One of the main focuses of this paper is on the efficient implementations for state-of-the-art GPUs, the implementation details of these algorithms will be discussed.

Considerable research has also been done on the impact of matrix reorderings and graph colorings on

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P. O. Box 808, L-561, Livermore, CA 94551 (li50@llnl.gov). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-788542).

†Department of Computer Science, College of Engineering and Computer Science, Arkansas State University, Jonesboro, AR 72401 (chaoyu.zhang@smail.astate.edu)

the performance of SpTrSV [21, 25, 36, 45]. In general, the performance of SpTrSV can be improved if the matrix has been permuted by graph colorings or by certain types of reorderings [25]. However, we remark that reordering or equivalently performing Gauss-Seidel iterations in different orders actually yields a different system to solve. Thus, coloring and reordering are not considered in this work. Another line of methods is based on the "partitioned inverses" proposed in [2, 3], where the inverse of a triangular matrix is represented as the product of multiple sparse triangular factors, and thus the solve reduces to matrix-vector products with the triangular matrix factors. Related works also include the approaches of using iterative methods to approximately solve sparse triangular systems for preconditioning purposes [5, 15]. These types of methods are also out of the scope of this work.

The remaining of the paper is organized as follows: a summary of the notations used in this paper is given in Section 2; the level-based and element-based scheduling approaches are reviewed in Sections 3 and 4 respectively; the CUDA implementations of the proposed algorithms are discussed in details in Section 5; numerical results of Gauss-Seidel iterations on structured and unstructured matrices are presented in Section 6 and finally we conclude in Section 7.

## 2 Notations

We consider the SpTrSV kernel of the form

$$(L+D)x = f \quad \text{or} \quad (U+D)x = f, \tag{2.1}$$

where sparse matrices $L \in \mathbb{R}^{n\times n}$ and $U \in \mathbb{R}^{n\times n}$ are strictly lower and upper triangular respectively, stored in the CSR format or the CSC format, $D$ is a diagonal matrix with no zero entries on the diagonal, saved in a vector $d$, and $f \in \mathbb{R}^n$ is the dense right-hand-side. Standard approaches for (2.1) are *forward and backward substitutions*.

## 3 Level-scheduling algorithms

Parallelism in SpTrSV can be discovered by analyzing the dependencies between unknowns. Unknown $x(i)$ can be immediately determined once all the other unknowns involved in equation $i$ become available. The dependencies can be analyzed by exploiting the underlying directed acyclic graph (DAG) associated with the triangular matrix. We associate the $(i, j)$ entry of the matrix if it is nonzero to the edge from node $j$ to node $i$ in the DAG, which indicates that the solution of $x(i)$ depends on that of $x(j)$. The idea is then to group the unknowns into different levels, where the first level consists of the nodes in the graph with zero in-degree and nodes in any level should only depend on those in the lower levels. Therefore, the system can be solved level by level and the unknowns within the same level can be computed simultaneously. This approach is often known as *level scheduling* [42]

**Algorithm 1** CSR SpTrSV with level scheduling

```
for m = 1, ..., nlev do
  parfor k = ilev(m), ..., ilev(m+1) − 1 do
    i := jlev(k)
    x(i) := f(i)
    for j = rowptr(i), ..., rowptr(i+1) − 1 do
      x(i) := x(i) − val(j) × x(colind(j))
    end for
    x(i) := x(i)/d(i)
  end parfor
end for
```

The levels of the unknowns can be easily obtained by exploiting a type of topological sorting of the DAG. The level of $x(i)$, denoted by $lev(i)$, can be simply computed by

**for** $i = 1, 2, \ldots, n$ **do**
  $lev(i) = 1 + \max\{lev(j)\}$, for $L_{ij} \neq 0$
**end for**

for the forward substitutions, where the matrix $L$ is accessed by rows. If the column access is more efficient, $lev(i)$ can be computed alternatively as

**for** $j = 1, 2, \ldots, n$ **do**
  $lev(i) = \max\{lev(j) + 1, lev(i)\}$, for $L_{ij} \neq 0$
**end for**

where all $lev(i)$ must be first initialized to zeros before the for-loop. The levels for the backward substitutions can be computed in the same way with the matrix $U$ and reversing the order of the loop.

SpTrSV with level scheduling in the CSR format is presented in Algorithm 1, where $nlev$ denotes the number of levels, $jlev$ is an array that lists the unknowns in a nondecreasing order of their levels, and array $ilev$ contains the pointers to the levels in $jlev$, i.e., $ilev(i)$ is the position in $jlev$ where level $i$ starts.

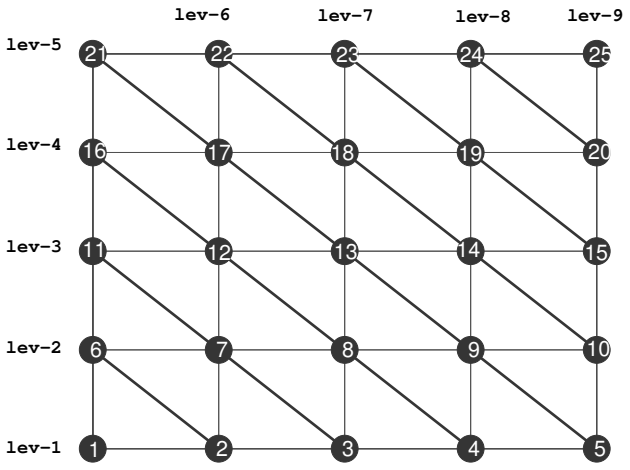
The level-scheduling algorithm with the CSC format is shown in Algorithm 2, where the solution $x$ should first be initialized to be equal to $f$. A remarkable difference in this algorithm is the requirement of a critical section around the concurrent updates to $x$, in order to avoid memory read and write conflicts in parallel.

Clearly, we have $1 \leq nlev \leq n$, and on average the degree of parallelism is $n/nlev$. The best scenario corresponds to the situation where all the unknowns can be computed simultaneously (i.e., for diagonal matrices, $nlev = 1$), whereas in the worst case, when $nlev = n$, each unknown is of a different level, so that the solve

**Algorithm 2** CSC SpTrSV with level scheduling

$x := f$
**for** $m = 1, \ldots, nlev$ **do**
  **parfor** $k = ilev(m), \ldots, ilev(m+1) - 1$ **do**
    $i := jlev\,(k)$
    $x(i) := x(i)/d(i)$
    **for** $j = colptr\,(i), \ldots, colptr\,(i+1) - 1$ **do**
      CRITICAL SECTION ENTRY
      $x(rowind(j)) := x(rowind(j)) - val(j) \times x(i)$
      CRITICAL SECTION EXIT
    **end for**
  **end parfor**
**end for**

Figure 1: Level scheduling for the 5-point stencil operator on a 2-D regular grid




becomes completely sequential. Thus, the performance of the level-scheduling algorithms significantly depends on the number of the levels. For a regular grid of size $n_x \times n_y \times n_z$ and the 7-point operator, we can write $nlev = n_x + n_y + n_z - 2$. In Figure 1, the levels in the $5 \times 5 \times 1$ grid are shown as an example.

## 4 Element-scheduling algorithms

The level-scheduling algorithms discussed in the previous section have a main drawback of including a global synchronization point between any two levels. It is not only that the synchronization itself represents an extra overhead but the synchronization can also stall jobs that are ready to go. In Figure 1, suppose that the computations on nodes 5, 9 and 13 are finished earlier than those on 17 and 21. By the level-scheduling method, the computations in level 6 will not be started until the entire level 5 finishes, although nodes 10 and 14 are free of dependencies immediately after the jobs on nodes 5, 9 and 13 are done. This issue motivated a more fine-grained element-scheduling approach that can schedule more aggressively the computations of the elements that are ready to be solved and can also avoid global synchronizations. It is worth pointing out that the element-scheduling approach has the same degree of parallelism as that in the level scheduling; however, it is the latency between the time when an unknown is ready to be solved and the time it is actually being solved that can be reduced.

SpTrSV with element scheduling in the CSR format is presented in Algorithm 3, where an array $ready(j)$ is used to indicate if $x(j)$ has already been solved, which should be initialized to be 0. Note that there is a busy-waiting loop for each individual dependency that iterates until the corresponding unknown is ready, and $x(i)$ is computed when all the unknowns that $i$ depends on become available. After $x(i)$ is computed, $ready(i)$ can be set to 1. Taking a closer look at this algorithm, compared with Algorithm 1, a finer-level concurrency can actually be achieved as well in the reduction-loop, i.e., "for $j = rowptr(i), \ldots, rowptr(i+1) - 1$", where individual partial sum $val(j) \times x(colind(j))$ can be computed as soon as $x(colind(j))$ is ready, which is indicated by $ready(colind(j))$, whereas in Algorithm 1, the whole reduction is not started until the solutions of all the dependencies are available and $x(i)$ is ready to be solved.

**Algorithm 3** CSR SpTrSV with element scheduling

**parfor** $i = 1, 2, \ldots, n$ **do**
  $x(i) := f(i)$
  **for** $j = rowptr(i), \ldots, rowptr(i+1) - 1$ **do**
    **while** $ready(colind(j)) = 0$ **do**
      {busy waiting}
    **end while**
    $x(i) := x(i) - val(j) \times x(colind(j))$
  **end for**
  $x(i) := x(i)/d(i)$
  $ready(i) := 1$
**end parfor**

SpTrSV with element scheduling in the CSC format is described in Algorithm 4, which uses a different scheme that is based on counters to resolve the dependencies. A counter $count(i)$ is maintained, for all $i = 1, \ldots, n$, to keep the number of dependencies of $i$ that have not been unfinished, which should be initialized as the number of nonzeros in the $i$-th row of the triangular matrix. Once $count(i)$ reaches zero, it follows that $x(i)$ is ready to be solved and after that the corresponding counters of all the unknowns that depend on $i$

should be decreased by 1. Note that this decrement operation should also be put inside the critical region since the decrements to the same $count(i)$ can be executed in parallel from different unknowns.

**Algorithm 4** CSC SpTrSV with element scheduling

$x := f$
**parfor** $i = 1, 2, \ldots, n$ **do**
  **while** $count(i) \neq 0$ **do**
    {do nothing}
  **end while**
  $x(i) := x(i)/d(i)$
  **for** $j = colptr(i), \ldots, colptr(i+1) - 1$ **do**
    CRITICAL SECTION ENTRY
    $x(rowind(j)) := x(rowind(j)) - val(j) \times x(i)$
    $count(rowind(j)) := count(rowind(j)) - 1$
    CRITICAL SECTION EXIT
  **end for**
**end parfor**

## 5 Implementation details

**5.1 SpTrSV kernels** The aforementioned SpTrSV algorithms have been implemented for NVIDIA GPUs, namely that are the level scheduling in the CSR format (LEVR), the level scheduling in the CSC format (LEVC), the element scheduling in the CSR format (ELMR), and the element scheduling in the CSC format (ELMC). In this section, we discuss their CUDA kernels in turn and also the implementations of the Gauss-Seidel iterations using the SpTrSV kernels. Only the forward substitutions will be presented, while the extension to the backward substitutions is straightforward.

We shall start with the implementation of kernel `LEVR`, which is corresponding the parfor-loop of Algorithm 1, where `l1` $= ilev(m)$ and `l2` $= ilev(m+1)$ for a given level $m$. A group of `K` threads, where `K` is assumed to be 2, 4, 8, 16 or 32, is dispatched for one unknown of level $m$ that is associated with the global index of the group, `gid`. The pointers of the corresponding row are loaded by the first two threads of the group and broadcast to all the threads in the group by the warp-level primitive, `__shfl_sync`. The dot-product between a matrix row and the right-hand side is performed by the parallel reduction with the group of threads using warp shuffles as well. Finally, the first thread of a group computes the result and saves it back to `x(i)`.

```
template <int K, typename T>
__global__ void
LEVR(T *f, T *x, T *d, T *val, int *colind,
     int *rowptr, int *jlev, int l1, int l2)
{
  int gid = (blockIdx.x * blockDim.x +
             threadIdx.x) / K;
  int lane = threadIdx.x & (K - 1);
  int p = 0, q = 0, i = -1;
  T sum = 0.0;
  if (gid + l1 < l2 && lane < 2) {
    i = jlev[gid+l1];
    p = rowptr[i+lane];
  }
  q = __shfl_sync(-1, p, 1, K);
  p = __shfl_sync(-1, p, 0, K);
  for (p += lane; p < q; p += K)
    sum += val[p] * x[colind[p]];
  #pragma unroll // parallel reduction
  for (int d = K/2; d > 0; d >>= 1)
    sum += __shfl_down_sync(-1, sum, d);
  if (i >= 0 && lane == 0)
    x[i] = (f[i] - sum) / d[i];
}
```

The implementation of kernel `LEVC` that corresponds to the parfor-loop of Algorithm 2 is shown below, where the input solution vector `x` is assumed to have been initialized as the right-hand side. To save memory transactions, we let only the first thread of a group read `x(i)` and `d(i)`, compute `x(i)`, and then broadcast the result to all the threads of the group. The solution updating associated with the most inner for-loop of Algorithm 2 is done in parallel by the group of threads, where the critical section is implemented by using the CUDA `atomicAdd` operation.

```
template <int K, typename T>
__global__ void
LEVC(T *x, T *d, T *val, int *rowind,
     int *colptr, int *jlev, int l1, int l2)
{
  int gid = (blockIdx.x * blockDim.x +
             threadIdx.x) / K;
  int lane = threadIdx.x & (K - 1);
  int p = 0, q = 0;
  T t = 0.0;
  if (gid + l1 < l2 && lane < 2) {
    int i = jlev[gid+l1];
    p = colptr[i+lane];
    if (lane == 0)
      x[i] = t = x[i] / d[i];
  }
  q = __shfl_sync(-1, p, 1, K);
  p = __shfl_sync(-1, p, 0, K);
  t = __shfl_sync(-1, t, 0, K);
  for (p += lane; p < q; p += K)
    atomicAdd(&x[rowind[p]], -t*val[p]);
}
```

The above two level-scheduling kernels solve the unknowns of a given level at a time, so that the entire substitutions can be done by repeatedly launching the kernel in an outer-loop shown as follows, in which way the global synchronization across thread blocks can be achieved. Furthermore, an optimization regarding reducing the kernel launching overhead proposed in [34]

is adopted, where the consecutive kernels that require only one thread block are grouped into a single kernel for a chain of levels, and inside the kernel, `__syncthreads()` is used to synchronize the threads of the block between the levels.

```
for (int m = 0; m < nlev; m++)
  LEVR<<<...>>>(x,d,val,colind,rowptr,jlev,
                ilev[m],ilev[m+]);
```

In the rest of this section, we discuss the implementations of SpTrSV with the element-scheduling approach. The following kernel `ELMR` implements Algorithm 3, where a fixed number of 32 threads (a warp) is used for each unknown and each thread keeps checking if the corresponding dependency has been ready. Once it is ready, its contribution is added to the partial sum. The value `f[i]` and the diagonal entry `d[i]` are prefetched by the first thread of a warp prior to the busy-waiting loop. When all the dependencies have been resolved, it follows that the reduction is done in parallel within the warp and the first thread computes and saves the result as in `LEVR`. Note here that a `__threadfence()` is put up after updating `x[i]` and before setting `ready[i]`. This is to guarantee that at the time when the threads working on the unknowns that depend on `x[i]` see its readiness, the correct result of `x[i]` should already have been observed by all the threads in the device. Moreover, adding the keyword `volatile` to a variable is to tell the compiler that this variable can be changed at any time by other threads and therefore any reference to this variable compiles to an actual memory read or write instruction [37].

```
template <typename T>
__global__ void
ELMR(int n, T *f, volatile T *x, T *d, T *val,
     int *colind, int *rowptr, int *jlev,
     volatile char *ready)
{
  int wid = (blockIdx.x * blockDim.x +
             threadIdx.x) >> 5;
  int lane = threadIdx.x & (warpSize - 1);
  if (wid >= n) return;
  int p = 0, q = 0, i = -1;
  T sum = 0.0, diag;
  if (lane < 2) {
    i = jlev[wid];
    p = rowptr[i+lane];
  }
  q = __shfl_sync(-1, p, 1);
  p = __shfl_sync(-1, p, 0);
  if (lane == 0) {
    sum = f[i];  diag = d[i];
  }
  for (p += warp_lane; p < q; p += warpSize) {
    while (ready[colind[p]] == 0);
    sum -= val[p] * x[colind[p]];
  }
  #pragma unroll // parallel reduction
  for (int d = warpSize/2; d > 0; d >>= 1)
    sum += __shfl_down_sync(-1, sum, d);
  if (lane == 0) {
    x[i] = sum / diag;
    __threadfence();
    ready[i] = 1;
  }
}
```

Finally, we discuss the implementation of kernel `ELMC` associated with Algorithm 4 that uses a counter-based busy-waiting scheme. The first thread of a warp constantly compares `count[i]` with zero in the busy-waiting loop, and computes `x[i]` afterwards when it goes out of the busy loop. The column-wise updating to the unknowns that depend on `x[i]` and the decrements of the corresponding counters are performed with the atomic functions, where a `__threadfence()` is required after each update to the solution and before decreasing the counter to makes sure that the counter decrement is only performed after the updated solution has been written to global memory.

```
template <typename T>
__global__ void
ELMC(int n, volatile T *x, T *d, T *val,
     int *rowind, int *colptr, int *jlev,
     volatile int *count)
{
  int wid = (blockIdx.x * blockDim.x +
             threadIdx.x) >> 5;
  int lane = threadIdx.x & (warpSize - 1);
  if (wid >= n) return;
  int p = 0, q = 0;
  T t = 0.0;
  if (lane < 2) {
    int i = jlev[wid];
    p = colptr[i+lane];
  }
  q = __shfl_sync(-1, p, 1);
  p = __shfl_sync(-1, p, 0);
  if (lane == 0) {
    t = 1.0 / d[i];
    while (count[i] != 0);
    x[i] = t = x[i] * t;
  }
  t = __shfl_sync(-1, t, 0);
  for (p += lane; p < q; p += warpSize) {
    atomicAdd(&x[rowind[p]], -t*val[p]);
    __threadfence();
    atomicSub(&count[rowind[p]], 1);
  }
}
```

Before closing this section, we remark that our ELMC algorithm was indeed inspired by the "global synchronization free" algorithm proposed in [27], where a scheduling approach similar to the element scheduling was used as well as the counter-based scheme to resolve dependencies. We list two major differences between the

two algorithms and their implementations as follows:

1. In [27], the warp that has the global warp index $i$ is dispatched to solve the unknown $x(i)$, whereas in our algorithm, the mapping between warps and the unknowns respects the levels of the unknowns, i.e., warp $i$ is dispatched to unknown $x(jlev(i))$. Note that as long as the assumption holds that the CUDA runtime scheduler activates the warps based on their global warp indices, i.e., a warp $i$ must be activated before a warp $j$ if $i < j$, both algorithms can be guaranteed to be deadlock-free.

2. In our implementations, only the first thread of a warp is spinning on the lock, whereas the other threads of the warp are waiting for the first thread to finish the busy-waiting loop before proceeding to the later instructions. This coordination is guaranteed by the warp-level synchronization, that is, a warp can only execute one common instruction at a time and warp divergence is serialized. On the other hand, in [27], the whole warp is spinning.

As will be shown later in Section 6, the above modifications could yield tremendous performance gains to the existing synchronization-free approach; however, on the negative side, the dispatching approach in our algorithms requires the level information, and consequently, the analysis phase is more expensive in order to compute the levels.

**5.2 Gauss-Seidel iterations** The Gauss-Seidel iterations are considered in this work as the application of the developed SpTrSV kernels. The forward Gauss-Seidel iterations can be represented in a matrix form

$$(L + D)x_{k+1} + Ux_k = f, \tag{5.2}$$

where the whole matrix $A = L + D + U$ is assumed to be given as the input matrix. The Gauss-Seidel kernels in the CSR format can be easily modified from the CSR format SpTrSV kernels, where, in general, the output $x_{k+1}$ and the input $x_k$ should be stored in two separate vectors and the statement in the inner for-loops of Algorithms 1 and 3, i.e.,

$$x(i) := x(i) - val(j) \times x(colind(j)),$$

should be changed to

**if** $colind(j) < i$ **then**
  $x_{k+1}(i) := x_{k+1}(i) - val(j) \times x_{k+1}(colind(j))$
**else if** $colind(j) > i$ **then**
  $x_{k+1}(i) := x_{k+1}(i) - val(j) \times x_k(colind(j))$
**end if**

to read appropriate values from $x_k$ and $x_{k+1}$ corresponding to the lower and the upper triangular part of row $i$. However, when $A$ is symmetric, it follows that $x_k$ and $x_{k+1}$ can share the same memory. In other words, (5.2) can be performed "in-place" with a single vector $x$ as the input and the output, which can be shown by checking the dependencies of unknowns as the following. For some $a_{i,j} \neq 0$ with $j > i$, we also have $a_{j,i} \neq 0$, so $x(j)$ depends on $x(i)$ in the forward sweep and the level of $x(j)$ must be higher than the one of $x(i)$. Hence, at the time of solving $x(i)$, the value of $x(j)$ must have not been changed from its original value in $x_k(j)$.

On the other hand, for executing the Gauss-Seidel iterations (5.2) using the CSC format SpTrSV kernels, we can compute a new right-hand side, $r_k := f - Ux_k$, followed by solving $(L + D)x_{k+1} = r_k$, or equivalently we can compute

$$r_k := f - Ax_k, \quad x_{k+1} = x_k + (L + D)^{-1}r_k, \tag{5.3}$$

in the cases where performing the matrix-vector product with a triangular part of $A$ is not readily available. All the above results can be easily extended to the backward Gauss-Seidel iterations.

**5.3 Analysis phases** All the parallel SpTrSV algorithms discussed in this paper require an analysis phase to computes the levels of the unknowns and, in addition, the dependency counts for the ELMC algorithm. The analysis phase may be considered as an extra overhead; however, several factors should be taken into account: First, the analysis is usually not very expensive relative to the solve, so a faster parallel solve can pay off the cost of the analysis. Second, more importantly, in many applications, such as the Gauss-Seidel iterations considered in this work, multiple solves are required with the same matrix, so the cost of the analysis phase can be amortized. Third, there also exist parallel topological sorting algorithms for determining the levels, proposed by Kahn [22] in 1962, which can further reduce the cost. The first CUDA implementation of Kahn's algorithm was due to Naumov [34] and later in [24] using a modified parallel breadth first search (BFS). In this work, the levels of the unknowns were determined with the simple approach discussed in Section 3 on the CPU. We refer to [24] for the performance of the analysis phase running on GPUs and its cost relative to the overall cost of the iterations.

## 6 Numerical experiments

The experiments were conducted on the machines at Lawrence Livermore National Laboratory, equipped with NVIDIA P100 and V100 GPUs and IBM POWER CPUs. The CUDA program was compiled by `nvcc` with option `-gencode arch=compute_60,"code=sm_60"` for P100, and `compute_70, sm_70` for V100. For the CUDA

kernel configurations, thread blocks are one-dimensional of size 512. Our SpTrSV kernels were used to implement the Gauss-Seidel iterations and compared with the two solvers `cusparseDcsrsv` and `cusparseDcsrsv2` available from cuSPARSE v9.2, denoted by CUS1 and CUS2 in the following of the paper, and the implementation of the global-synchronization-free algorithm available from bhSPARSE [28] denoted by GSF. The `cusparseDcsrsv` solver adopted a similar level-scheduling approach [34], whereas the algorithm used in `cusparseDcsrsv2` has not been published.

In the following of this section, we will show the performance on structured matrices obtained from discretized Laplace operators, and some general matrices. All the tests were performed in double precision. For each test matrix, we performed a forward Gauss-Seidel sweep followed by a backward one, the performance of which was measured in GFLOPS given by

$$\text{GFLOPS} = \frac{4 \times nnz - 2 \times n}{t \times 10^9}, \tag{6.4}$$

where we denote by $nnz$ the number of nonzeros of the matrix and by $t$ the time measured in seconds. The CSR format Gauss-Seidel kernels were modified from the corresponding SpTrSV kernels, and the CSC format Gauss-Seidel sweeps were performed in two steps as in (5.3), where the matrix-vector product was computed with $A$ in the CSR format for better performance. The timings of the analysis phase are omitted.

**6.1 Laplacian matrices** We shall start our performance study with a set of discretized Laplacians obtained from the finite-difference discretization on regular grids of dimension $n_x \times n_y \times n_z$, where the numbers of the mesh points keep the same, while the aspect ratios, $\rho_{yx} = n_y/n_x$ for 2-D grids and $\rho_{zx} = n_z/n_x$ for 3-D grids, vary. Since the number of the levels in SpTrSV increases with the aspect ratios and thus the degree of parallelism decreases, lower performance is expected for grids with higher aspect ratios. For the 2-D grids, we tested Laplacian matrices with standard 5- and 9- point stencils, while for the 3-D grids we tested the standard 7- and 27-point Laplacian matrices.

In Table 1, we show the performance of the Gauss-Seidel sweeps with the 7 different SpTrSV kernels for the 2-D Laplacian matrices on P100 and V100 GPUs. For most of the matrices, our level-scheduling based kernels `LEVR` and `LEVC` outperformed their counterpart `CUS1` in cuSPARSE, whereas the only exception was on V100 with the 9-point operator on the last grid, where `CUS1` was considerably faster than `LEVR` and `LEVC`. In addition, the performances of `LEVR` and `LEVC` were very close for all the cases. On the other hand, our element-scheduling based kernels, `ELMR` and `ELMC`, ran much faster than the other kernels, and, moreover, `ELMR` exhibited better performance than `ELMC` in general.

Table 1: Performance of forward and backward Gauss-Seidel sweeps for 2-D Laplacians (in GFLOPS) with the best results highlighted in boldface.

(a) 5-point stencil on P100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 2048 × 2048 | 2.11 | 2.27 | 4.59 | **4.76** | 1.75 | 3.16 | 0.02 |
| 1024 × 4096 | 1.79 | 1.95 | **4.50** | 4.07 | 1.41 | 3.43 | 0.04 |
| 512 × 8192 | 1.10 | 1.15 | **3.23** | 2.70 | 0.86 | 2.29 | 0.08 |
| 256 × 16384 | 0.59 | 0.60 | **1.75** | 1.61 | 0.46 | 1.33 | 0.15 |
| 128 × 32768 | 0.30 | 0.30 | **0.94** | **0.94** | 0.24 | 0.73 | 0.27 |

(b) 9-point stencil on P100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 2048 × 2048 | 2.79 | 2.90 | **7.12** | 6.17 | 2.13 | 5.12 | 0.04 |
| 1024 × 4096 | 1.93 | 1.96 | **5.41** | 4.52 | 1.48 | 3.77 | 0.07 |
| 512 × 8192 | 1.04 | 1.05 | **3.05** | 2.76 | 0.83 | 2.15 | 0.13 |
| 256 × 16384 | 0.54 | 0.54 | **1.59** | 1.58 | 0.43 | 1.20 | 0.25 |
| 128 × 32768 | 0.27 | 0.27 | 0.71 | **0.86** | 0.31 | 0.49 | 0.42 |

(c) 5-point stencil on V100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 2048 × 2048 | 2.07 | 2.09 | **6.10** | 5.25 | 2.10 | 2.07 | 0.04 |
| 1024 × 4096 | 1.60 | 1.63 | **5.44** | 4.31 | 1.70 | 1.85 | 0.06 |
| 512 × 8192 | 0.97 | 0.98 | **3.56** | 2.49 | 1.07 | 1.04 | 0.12 |
| 256 × 16384 | 0.49 | 0.50 | **1.90** | 1.11 | 0.51 | 0.42 | 0.24 |
| 128 × 32768 | 0.25 | 0.26 | **0.94** | 0.51 | 0.23 | 0.14 | 0.42 |

(d) 9-point stencil on V100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 2048 × 2048 | 2.50 | 2.55 | **8.50** | 6.08 | 2.71 | 2.09 | 0.08 |
| 1024 × 4096 | 1.60 | 1.64 | **5.99** | 4.11 | 1.53 | 0.99 | 0.14 |
| 512 × 8192 | 0.90 | 0.91 | **3.32** | 1.92 | 0.88 | 0.27 | 0.28 |
| 256 × 16384 | 0.46 | 0.47 | **1.69** | 0.78 | 0.42 | 0.09 | 0.53 |
| 128 × 32768 | 0.22 | 0.23 | **0.66** | 0.56 | 0.49 | 0.08 | **0.66** |

As expected, the performance of most of the kernels steadily degraded when the aspect ratio $n_y/n_x$ increased as the number of levels were increased correspondingly. However, the kernel `GSF`, on the contrary, behaved very differently, which performed exceedingly poorly for the first grids with lower aspect ratios but eventually became competitive again for the last grid. We remark here that the dramatic performance improvement to `GSF` by our element-scheduling based kernels was mainly from the use of the information of the levels. By and large, the speedup of a factor 1.4 was achieved by `ELMR` over cuSPARSE on P100, and a factor of up to 3.2 was achieved on V100.

The performance for the 3-D Laplacian matrices are presented in Table 2, where, compared with the 2-D problems, the GFLOPS numbers are much higher since there are much more parallelism can be exploited due

Table 2: Performance of forward and backward Gauss-Seidel sweeps for 3-D Laplacians (in GFLOPS) with the best results highlighted in boldface.

(a) 7-point stencil on P100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 128 × 128 × 128 | 4.97 | 5.10 | 5.23 | **5.53** | 5.02 | 4.80 | 0.35 |
| 64 × 128 × 256 | 4.91 | 5.13 | 5.82 | **5.85** | 5.04 | 5.03 | 0.39 |
| 64 × 64 × 512 | 4.64 | 4.93 | 6.80 | **7.83** | 4.95 | 5.66 | 0.44 |
| 32 × 64 × 1024 | 3.54 | 3.81 | 7.05 | **8.26** | 3.94 | 5.77 | 0.45 |
| 32 × 32 × 2048 | 2.91 | 3.16 | **6.86** | 5.96 | 2.32 | 5.09 | 0.59 |

(b) 27-point stencil on P100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 128 × 128 × 128 | 13.83 | 13.97 | 17.91 | **19.06** | 9.85 | 15.06 | 1.21 |
| 64 × 128 × 256 | 12.27 | 12.08 | **21.51** | 21.40 | 8.98 | 15.96 | 0.91 |
| 64 × 64 × 512 | 9.93 | 10.17 | **22.29** | 16.82 | 6.64 | 13.54 | 1.17 |
| 32 × 64 × 1024 | 5.66 | 6.11 | **11.21** | 10.92 | 4.16 | 6.46 | 1.10 |
| 32 × 32 × 2048 | 2.97 | 3.02 | 5.53 | **6.58** | 2.22 | 3.18 | 1.56 |

(c) 7-point stencil on V100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 128 × 128 × 128 | 6.88 | 7.32 | 8.51 | **9.03** | 7.79 | 8.06 | 0.74 |
| 64 × 128 × 256 | 7.11 | 7.69 | 9.66 | **10.90** | 8.45 | 8.81 | 0.73 |
| 64 × 64 × 512 | 6.71 | 7.05 | 10.79 | **13.87** | 8.02 | 10.43 | 0.84 |
| 32 × 64 × 1024 | 5.16 | 5.19 | 10.79 | **11.62** | 4.76 | 8.51 | 0.86 |
| 32 × 32 × 2048 | 2.62 | 2.67 | **8.45** | 6.51 | 2.51 | 4.76 | 1.18 |

(d) 27-point stencil on V100

| Grid | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 128 × 128 × 128 | 19.53 | 18.74 | **34.57** | 30.33 | 16.87 | 28.40 | 1.91 |
| 64 × 128 × 256 | 16.27 | 16.38 | **35.12** | 27.27 | 13.87 | 25.21 | 1.64 |
| 64 × 64 × 512 | 9.98 | 10.36 | **29.20** | 18.61 | 9.46 | 16.97 | 2.02 |
| 32 × 64 × 1024 | 5.01 | 5.16 | **14.27** | 9.97 | 4.79 | 7.17 | 1.96 |
| 32 × 32 × 2048 | 2.62 | 2.67 | **6.49** | 4.77 | 2.41 | 3.11 | 2.86 |

to the fact that the numbers of levels are much fewer and also there are more operations for each unknown from the larger stencil sizes. When comparing among the kernels, the results are similar as those in the 2-D cases, where `ELMR` and `ELMC` remained the fastest for all the problems, while, however, there were several cases that `ELMC` was actually faster than `ELMR`. The speedups of factors up to 1.7 and 2.0 relative to cuSPARSE were obtained on P100 and V100 respectively by the two element-scheduling kernels. The throughput of `GSF` consistently stayed low compared with the other kernels.

Furthermore, it is worth mentioning that for the 3-D problems, the performance of running all the kernels on nearly all the grids on V100 was considerably higher than that on P100, which was, in contrast, often not the case for running the 2-D problems.

**6.2 General matrices** In this section, we report the experiment results on 19 matrices selected from the SuiteSparse Matrix Collection [17] and 2 matrices from the linear elasticity problems discretized by finite element methods on unstructured meshes using MFEM [1]. The dimension (N), the number of the nonzeros (NNZ), the average number of non-zeros per row (RNZ), the number of levels in the $L$ and $U$ parts of the matrices (NLEV = NLEV($L$) + NLEV($U$)) and the average degree of parallelism with respect to the number of unknowns (NPAR = 2N/NLEV) of each matrix are tabulated in Table 3, and short descriptions of the test matrices are given in Table 4. The sizes of the test matrices range from several hundred of thousands to a few million, and the densities of the matrices vary from a few nonzeros per row to several scores. The average degrees of parallelism also have significant variations.

Table 3: The name and a short description of each test matrix.

| Matrix | DESCRIPTION | Matrix | DESCRIPTION |
|---|---|---|---|
| 3D_thermal2 | thermal problem | elasticity2D | 2D FEM elasticity |
| thermomech | thermal problem | elasticity3D | 3D FEM elasticity |
| offshore | electromagnetics | thermal2 | steady state thermal |
| ASIC_320ks | circuit Simulation | atmosmodd | atmospheric model |
| cage13 | DNA electrophoresis | StocF-1465 | flow in porous media |
| af_shell3 | sheet metal forming | af_shell10 | sheet metal forming |
| af_shell8 | sheet metal forming | G3_circuit | circuit Simulation |
| parabolic_fem | convection diffusion | Transport | flow and transport |
| apache2 | structural problem | Bump_2911 | reservoir simulation |
| ecology2 | landscape ecology | Queen_4147 | 3D structural |
| webbase1M | web connectivity | | |

In Table 5, we report the performance of the Gauss-Seidel iterations for the general matrices on P100 and V100. From the results, we can see that high GFLOPS numbers were achieved by the matrices with both large average degrees of parallelism and large average numbers of nonzeros per row. On the whole, the ranking of the performance among test matrices, measured in GFLOPS, actually matches the ordering by the product of RNZ and NPAR very well. Specifically, the matrices that have the 4 largest values of RNZ × NPAR, namely `parabolic_fem`, `cage13`, `elasticity3D` and `Transport`, yielded the highest throughput among all the matrices, while, on the other hand, the matrices that have the smallest values yielded poor performance, such as `3D_thermal2`, `offshore` and `ecology2`.

Comparing the different kernels, our two element-scheduling kernels, by and large, turned out to have the best performance, where `ELMC` was the fastest for 12 out of the 21 matrices on P100 and for 4 matrices on V100, while `ELMR` won for 5 matrices on P100 and for 15 matrices on V100. There were still a few cases where `LEVR` and `LEVC` could work better, including the matrix `parabolic_fem` that is of the size about half million and only has 14 levels, which makes the level-scheduling approach very efficient. Comparing with `CUS1` and `CUS2`

Table 4: The order (N), the number of the nonzeros (NNZ), the average number of nonzeros per row (RNZ), the number of levels (NLEV), and the average degree of parallelism (NPAR) of each test matrix.

| Matrix | N | NNZ | RNZ | NLEV | NPAR |
|---|---|---|---|---|---|
| 3D_thermal2 | 147,900 | 3,489,300 | 23.6 | 7,514 | 39 |
| thermomech | 204,316 | 2,846,228 | 13.9 | 1288 | 317 |
| offshore | 259,789 | 4,242,673 | 16.3 | 6,904 | 75 |
| ASIC_320ks | 321,671 | 1,316,085 | 4.1 | 70 | 9,191 |
| cage13 | 445,315 | 7,479,343 | 16.8 | 166 | 5,365 |
| af_shell3 | 504,855 | 17,562,051 | 34.8 | 7,450 | 136 |
| af_shell8 | 504,855 | 17,588,875 | 34.8 | 7,450 | 136 |
| parabolic_fem | 525,825 | 3,674,625 | 7.0 | 14 | 75,118 |
| apache2 | 715,176 | 4,817,870 | 6.8 | 1,328 | 1,077 |
| ecology2 | 999,999 | 4,995,991 | 5.0 | 3,998 | 500 |
| webbase1M | 1,000,005 | 3,105,536 | 3.1 | 1,026 | 1,449 |
| elasticity2D | 1,002,528 | 14,012,744 | 14.0 | 5,660 | 354 |
| elasticity3D | 1,029,000 | 80,990,208 | 78.7 | 2,904 | 709 |
| thermal2 | 1,228,045 | 8,580,313 | 7.0 | 2,478 | 991 |
| atmosmodd | 1,270,432 | 8,814,880 | 6.9 | 704 | 3,609 |
| StocF-1465 | 1,465,137 | 21,005,389 | 14.3 | 6,008 | 488 |
| af_shell10 | 1,508,065 | 52,672,325 | 34.9 | 16,720 | 180 |
| G3_circuit | 1,585,478 | 7,660,826 | 4.8 | 5,188 | 611 |
| Transport | 1,602,111 | 23,500,731 | 14.7 | 1,142 | 2,806 |
| Bump_2911 | 2,911,419 | 127,729,899 | 43.8 | 20,548 | 283 |
| Queen_4147 | 4,147,110 | 329,499,284 | 79.4 | 24,228 | 342 |

Table 5: Performance of forward and backward Gauss-Seidel sweeps for general matrices (in GFLOPS) with the best results highlighted in boldface.

(a) P100

| Matrix | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 3D_thermal2 | 0.55 | 0.62 | 0.96 | **1.45** | 0.34 | 0.60 | 0.65 |
| thermomech | 2.99 | 3.12 | 3.46 | **5.23** | 1.89 | 4.58 | 3.56 |
| offshore | 0.62 | 0.76 | 1.32 | **1.91** | 0.44 | 0.82 | 0.72 |
| ASIC_320ks | 4.68 | **5.87** | 5.05 | 5.29 | 5.01 | 4.19 | 2.86 |
| cage13 | **14.51** | 13.90 | 12.22 | 13.06 | 9.95 | 10.50 | 5.38 |
| af_shell3 | 1.94 | 2.33 | 4.08 | **6.73** | 1.63 | 2.36 | 0.22 |
| af_shell8 | 1.95 | 2.34 | 4.09 | **6.74** | 1.63 | 2.38 | 0.22 |
| parabolic_fem | **19.36** | 13.95 | 6.52 | 6.94 | 11.54 | 5.66 | 5.39 |
| apache2 | 3.18 | 3.19 | **6.19** | 6.12 | 2.44 | 4.80 | 0.17 |
| ecology2 | 1.13 | 1.24 | **3.28** | 2.86 | 0.91 | 2.28 | 0.04 |
| webbase1M | 1.13 | 1.66 | 1.81 | **2.23** | 1.37 | 1.69 | 1.91 |
| elasticity2D | 2.40 | 2.53 | **6.60** | 5.83 | 1.76 | 4.45 | 0.08 |
| elasticity3D | 16.87 | 16.99 | **28.24** | 25.39 | 12.49 | 19.47 | 1.75 |
| thermal2 | 2.77 | 3.18 | **4.90** | **4.90** | 2.18 | 4.11 | 1.63 |
| atmosmodd | **7.25** | 6.40 | 5.68 | 6.16 | 4.82 | 5.15 | 0.31 |
| StocF-1465 | 3.16 | 3.50 | 6.64 | **7.54** | 2.26 | 5.32 | 1.33 |
| af_shell10 | 2.64 | 3.17 | 4.20 | **8.07** | 2.15 | 2.69 | 0.12 |
| G3_circuit | 1.32 | 1.42 | **3.73** | 3.25 | 1.06 | 2.79 | 0.16 |
| Transport | 10.79 | 10.16 | 11.46 | **12.21** | 7.59 | 9.81 | 0.57 |
| Bump_2911 | 4.80 | 5.80 | 9.19 | **12.84** | 3.87 | 7.47 | 1.26 |
| Queen_4147 | 8.75 | 10.23 | 15.47 | **18.38** | 6.70 | 13.95 | 1.25 |

(b) V100

| Matrix | LEVR | LEVC | ELMR | ELMC | CUS1 | CUS2 | GSF |
|---|---|---|---|---|---|---|---|
| 3D_thermal2 | 0.68 | 0.76 | **0.98** | 0.79 | 0.34 | 0.34 | 0.91 |
| thermomech | 3.87 | 3.85 | 4.71 | 4.60 | 3.00 | **5.21** | 5.09 |
| offshore | 0.67 | 0.80 | **1.36** | 1.04 | 0.51 | 0.77 | 1.04 |
| ASIC_320ks | 5.98 | 7.84 | 7.68 | **8.78** | 7.34 | 8.05 | 4.76 |
| cage13 | 21.67 | 20.49 | 23.18 | **23.37** | 16.89 | 19.44 | 9.73 |
| af_shell3 | 1.75 | 1.80 | **5.33** | 3.99 | 1.79 | 3.39 | 0.37 |
| af_shell8 | 1.78 | 1.80 | **5.35** | 3.98 | 1.79 | 3.36 | 0.37 |
| parabolic_fem | **26.84** | 20.52 | 9.83 | 11.93 | 20.48 | 12.08 | 10.93 |
| apache2 | 2.97 | 2.95 | **8.28** | 6.79 | 2.71 | 3.36 | 0.33 |
| ecology2 | 1.14 | 1.15 | **3.62** | 2.47 | 1.04 | 0.64 | 0.07 |
| webbase1M | 1.30 | 1.99 | 2.81 | **3.06** | 1.98 | 2.95 | 2.54 |
| elasticity2D | 2.08 | 2.17 | **7.83** | 4.65 | 2.04 | 1.80 | 0.16 |
| elasticity3D | 20.60 | 20.66 | **41.45** | 30.94 | 17.27 | 27.37 | 2.55 |
| thermal2 | 3.09 | 3.51 | **6.62** | 5.67 | 2.86 | 4.99 | 2.47 |
| atmosmodd | 9.62 | 8.69 | 9.16 | **10.91** | 7.86 | 8.15 | 0.64 |
| StocF-1465 | 3.13 | 3.72 | **8.62** | 6.59 | 3.03 | 5.26 | 2.28 |
| af_shell10 | 2.71 | 2.79 | **6.87** | 5.58 | 2.44 | 4.18 | 0.20 |
| G3_circuit | 1.35 | 1.37 | **4.43** | 3.20 | 1.21 | 2.17 | 0.26 |
| Transport | 14.79 | 12.71 | **21.00** | 20.33 | 11.54 | 16.66 | 1.10 |
| Bump_2911 | 5.18 | 5.51 | **11.80** | 11.37 | 5.28 | 9.22 | 2.18 |
| Queen_4147 | 10.02 | 10.89 | **21.99** | 18.81 | 9.14 | 17.74 | 1.83 |

from cuSPARSE and `GSF`, our proposed kernels achieved speedups for almost all the cases by a factor of up to 3.0 and on average 1.7 on P100, and the speedup by a factor of up to 3.8 and on average 1.6 on V100. The only exception that our kernels were not the fastest one was for the matrix `thermomech` on V100, where our best kernel `ELMR` was about 10% slower than `CUS2`, which was the fastest. Lastly, for many of the matrices, the performance of `GSF` was not very competitive compared with the other kernels.

When comparing the performance on the two types of the GPUs, we found that on V100 `ELMR`, `CUS1` and `GSF` were able to run a lot faster for nearly all the cases than running on P100, whereas `LEVR` and `LEVC` could be slightly slower for several cases, and moreover, the performance of `ELMC` and `CUS2` on V100 was found to deteriorate more severely for more cases. This performance issue on V100 is currently under the investigation by the authors.

## 7 Conclusions

In this paper, we considered the parallel algorithms for solving sparse triangular linear systems that are appropriate for the state-of-the-art GPUs. The proposed algorithms are based on different scheduling approaches and different matrix storage formats, in which several modifications to the existing approaches have been introduced that turned out to be able to greatly improve the overall performance. Furthermore, the efficient implementations in CUDA of the proposed algorithms have been carefully examined. As the application, the Gauss-Seidel iterations were implemented using the developed SpTrSV kernels in modified forms.

Numerical results on structured and unstructured matrices demonstrated the efficiency of the proposed algorithms and their implementations compared with state-of-the-art software packages, where significant performance improvements have been achieved on NVIDIA Pascal and Volta GPUs. Another practice includes applying ILU-type preconditioners with the Krylov subspace methods, where, in particular, the proposed implementations in both the CSR and the CSC format are useful for applying incomplete Cholesky or LDL factorization preconditioners, where only one triangular factor is usually stored. It is also worth mentioning that the performance of the proposed algorithms in general deteriorates with the amount of the fill-in in the ILU factorizations. Therefore, to achieve the best overall performance of ILU-preconditioned iterations, the users need to trade-off the speed of the convergence and the speed of solving the triangular systems when deciding the level of fill-ins to be introduced.

All the parallel sparse triangular solve algorithms considered in this paper require an analysis phase in order to generate the information to exploit the parallelism in the following solve phase. The justification of paying the extra cost in the analysis phase but having a faster parallel solve phase was provided for the scenarios of several important applications. In this work, the analysis phase remained executed on the CPU using a simple sequential algorithm. Parallel algorithms for the analysis phase on GPUs have been explored in [24, 34]. Multi-GPU versions of the level-based algorithms seem promising, where the unknowns within the same level can still be solved simultaneously after communicating the required solutions on other GPUs. However, on the other hand, extending the element-scheduling algorithms to distributed memory environment can be much more difficult.

### Acknowledgment

The first author would like to acknowledge the fruitful discussions with Professor Weifeng Liu on the global-synchronization-free SpTrSV algorithms in [27].

### Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

### References

[1] *MFEM: Modular finite element methods library.* `https://mfem.org`.

[2] F. L. Alvarado, A. Pothen, and R. Schreiber, *Highly Parallel Sparse Triangular Solution*, Springer New York, New York, NY, 1993, pp. 141–157.

[3] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*, SIAM Journal on Scientific Computing, 14 (1993), pp. 446–460.

[4] E. Anderson and Y. Saad, *Solving sparse triangular linear systems on parallel computers*, Int. J. High Speed Comput., 1 (1989), pp. 73–95.

[5] H. Anzt, E. Chow, and J. Dongarra, *Iterative Sparse Triangular Solves for Preconditioning*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 650–661.

[6] H. Anzt, S. Tomov, and J. Dongarra, *Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product*, in Proceedings of the Symposium on High Performance Computing, HPC '15, San Diego, CA, USA, 2015, Society for Computer Simulation International, pp. 75–82.

[7] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, *Fast sparse matrix-vector multiplication on GPUs for graph applications*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, Piscataway, NJ, USA, 2014, IEEE Press, pp. 781–792.

[8] J. L. Aurentz, V. Kalantzis, and Y. Saad, *Cucheb: A GPU implementation of the filtered Lanczos procedure*, Computer Physics Communications, 220 (2017), pp. 332 – 340.

[9] M. M. Baskaran and R. Bordawekar, *Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies*, IBM Reserach Report, RC24704 (W0812-047), (2008).

[10] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing, 34 (2012), pp. C123–C152.

[11] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High

Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 18:1–18:11.

[12] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM Trans. Graph., 22 (2003), pp. 917–924.

[13] L. Buatois, G. Caumon, and B. Lévy, *Concurrent number cruncher: a GPU implementation of a general sparse linear solver*, International Journal of Parallel, Emergent and Distributed Systems, 24 (2009), pp. 205–223.

[14] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, SIGPLAN Not., 45 (2010), pp. 115–126.

[15] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM Journal on Scientific Computing, 37 (2015), pp. C169–C193.

[16] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, *Solving lattice qcd systems of equations using mixed precision solvers on GPUs*, Computer Physics Communications, 181 (2010), pp. 1517 – 1528.

[17] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

[18] A. Dziekonski, M. Rewienski, P. Sypek, A. Lamecki, and M. Mrozowski, *GPU-accelerated LOBPCG method with inexact null-space filtering for solving generalized eigenvalue problems in computational electromagnetics analysis with higher-order fem*, Communications in Computational Physics, 22 (2017), p. 997–1014.

[19] R. Gandham, K. Esler, and Y. Zhang, *A GPU accelerated aggregation algebraic multigrid method*, Computers & Mathematics with Applications, 68 (2014), pp. 1151 – 1160.

[20] A. George, M. T. Heath, J. Liu, and E. Ng, *Solution of sparse positive definite systems on a shared-memory multiprocessor*, International Journal of Parallel Programming, 15 (1986), pp. 309–325.

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

[22] A. B. Kahn, *Topological sorting of large networks*, Commun. ACM, 5 (1962), pp. 558–562.

[23] J. Krüger and R. Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Trans. Graph., 22 (2003), pp. 908–916.

[24] R. Li, *On parallel solution of sparse triangular linear systems in CUDA*, CoRR, abs/1710.04985 (2017).

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

[26] R. Li, Y. Xi, L. Erlandson, and Y. Saad, *The eigenvalues slicing library (EVSL): Algorithms, implementation, and software*, SIAM Journal on Scientific Computing, 41 (2019), pp. C393–C415.

[27] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, *A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves*, Springer International Publishing, 2016, pp. 617–630.

[28] ———, *bhSPARSE: Benchmark SpTRSM using CSC*. `https://github.com/bhSPARSE/Benchmark_SpTRSM_using_CSC`, 2017.

[29] ———, *Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides*, Concurrency and Computation: Practice and Experience, (2017).

[30] W. Liu and B. Vinter, *An efficient GPU general sparse matrix-matrix multiplication for irregular data*, in Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, Washington, DC, USA, 2014, IEEE Computer Society, pp. 370–381.

[31] ———, *CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication*, in Proceedings of the 29th ACM International Conference on Supercomputing, ICS '15, New York, NY, USA, 2015, ACM, pp. 339–350.

[32] ———, *Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors*, Parallel Computing, 49 (2015), pp. 179 – 193.

[33] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, *Sparse matrix-matrix multiplication on modern architectures*, in 2012 19th International Conference on High Performance Computing, Dec 2012, pp. 1–10.

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

[35] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM Journal on Scientific Computing, 37 (2015), pp. S602–S626.

[36] M. Naumov, P. Castonguay, and J. Cohen, *Parallel graph coloring with applications to the incomplete-LU factorization on the GPU*, NVIDIA White Paper, (2015).

[37] NVIDIA, *CUDA C Programming Guide 10.1*, August 2019.

[38] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides, *Balancing locality and concurrency: Solving sparse triangular systems on GPUs*, in 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), Dec 2016, pp. 183–192.

[39] S. C. Rennich, D. Stosic, and T. A. Davis, *Accelerating sparse Cholesky factorization on GPUs*, Parallel Computing, 59 (2016), pp. 140 – 150. Theory and Practice of Irregular Applications.

[40] C. Richter, S. Schöps, and M. Clemens, *GPU acceleration of algebraic multigrid preconditioners for discrete elliptic field problems*, IEEE Transactions on Magnetics, 50 (2014), pp. 461–464.
[41] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr, *ViennaCL—linear algebra library for multi- and many-core architectures*, SIAM Journal on Scientific Computing, 38 (2016), pp. S412–S439.
[42] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd edition*, SIAM, Philadelpha, PA, 2003.
[43] J. H. Saltz, *Aggregation methods for solving sparse triangular systems on multiprocessors*, SIAM Journal on Scientific and Statistical Computing, 11 (1990), pp. 123–144.
[44] P. Sao, R. Vuduc, and X. S. Li, *A Distributed CPU-GPU Sparse Direct Solver*, Springer International Publishing, Cham, 2014, pp. 487–498.
[45] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, *Adapting sparse triangular solution to GPUs*, in 2012 41st International Conference on Parallel Processing Workshops, Sept 2012, pp. 140–148.
[46] M. Wang, H. Klie, M. Parashar, and H. Sudan, *Solving Sparse Linear Systems on NVIDIA Tesla GPUs*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 864–873.